Datamaster Manual

CarpeDM
Programming language for the DataMaster

Version	0.1.10
Last updated	December 2nd, 2020

Author	Mathias Kreider
Department	ACO
Group	TOS
Contact	m.kreider@gsi.de

DRAFT

Contents

1 Introduction **2**
- 1.1 General Machine Timing . . . 2
  - 1.1.1 New vs Old Design Philosophy . . . 3
  - 1.1.2 Responsibilities . . . 6
- 1.2 Timing System and the Datamaster . . . 8
  - 1.2.1 Control System Stack . . . 9
  - 1.2.2 Building Blocks . . . 9
- 1.3 Language for Accelerator Control . . . 11
  - 1.3.1 Schedule Graphs . . . 11
  - 1.3.2 Look, feel and function . . . 12
  - 1.3.3 Graph translation for an embedded system . . . 13

2 CarpeDM User Guide **15**
- 2.1 Getting Started . . . 16
  - 2.1.1 Installing carpeDM Tools . . . 16
  - 2.1.2 Installing Visualisation . . . 16
  - 2.1.3 Xdot Viewer Short Manual . . . 18
  - 2.1.4 Hello World! . . . 19
- 2.2 Command line tools . . . 21
- 2.3 Schedules . . . 21
  - 2.3.1 Overview . . . 21
  - 2.3.2 Basics . . . 22
- 2.4 Flow Control . . . 23
  - 2.4.1 Blocks and Changes during Runtime . . . 23
  - 2.4.2 Branches . . . 25
  - 2.4.3 Loops . . . 26
  - 2.4.4 Default Pattern Example . . . 28
- 2.5 Static Commands . . . 30
  - 2.5.1 Concept . . . 30
  - 2.5.2 Access Management . . . 30
  - 2.5.3 Counter Loop Example . . . 32

DRAFT

2.5.4 Timeout Loop Example 33
2.5.5 Alternation with Default Pattern Example 34

3 Offline Schedule Validation **35**
3.1 Schedule Structure Validation 35
3.1.1 Validation on creation 35
3.1.2 Validation on change 38
3.1.3 Intentional late message generation 39
3.1.4 Summary 39

4 Online Schedule Modification and Safeguards **40**
4.1 Overview 40
4.1.1 Problem Definition 40
4.1.2 Possible Approaches 41
4.2 Equivalent Static Model 43
4.2.1 Path Analyses 44
4.2.2 Orphaned command handling 45
4.2.3 Visual Reports 45
4.3 Enhanced Equivalent Static Model (aka “crystal ball”) 46
4.3.1 Problem Definition 46
4.3.2 Contextual inconsequence of default successors 46
4.3.3 Finding contributing optimisations 47
4.3.4 Covenants 48
4.3.5 Handling cursor to queue race conditions 49
4.4 Summary 49

5 Offline Resource Analysis **51**
5.1 Memory Load 51
5.1.1 Problem Definition 51
5.1.2 Graph Data 51
5.1.3 Meta Data 51
5.1.4 Load Balancing 52
5.1.5 Summary 52
5.2 Network Traffic 52
5.2.1 Problem Definition 52
5.2.2 Introduction to DNC 53
5.2.3 Introduction to DISCO DNC 53
5.2.4 DM to Endpoint Model 53
5.2.5 Arrival Curves from Schedules 53
5.2.6 Verification Process 53

DRAFT

5.2.7 Load Balancing 53
5.2.8 Summary 53

6 Theoretical Model **54**
6.1 Overview 54
6.1.1 Motivation 54
6.1.2 Choice of Implementation 54
6.2 Introduction to Network Calculus 55
6.2.1 Overview 55
6.2.2 Network Calculus Core Concepts 57
6.2.3 Mathematical Background 59
6.2.4 Elementary Building Blocks 63
6.2.5 Delay Analysis Methodology 68
6.3 Approach for modelling the Data Master 70
6.3.1 Overview 70
6.3.2 Machine Schedules as Flows 71
6.3.3 Outside Interference 74
6.3.4 Recurring Analyses 75
6.4 Scheduler Models 75
6.4.1 Scheduling under Network Calculus 76
6.4.2 Soft-CPU Scheduler 76
6.4.3 Processor Output 79
6.4.4 Priority Queue Scheduler 79
6.5 Etherbone Master – Framer 81
6.5.1 Etherbone Master Functional Recap 81
6.5.2 Input Parser 81
6.5.3 Header Generation 82
6.5.4 Output Flow and Service Curves 83
6.6 Etherbone Master – TX 83
6.6.1 Variable Length Function 84
6.6.2 Header 85
6.6.3 Finding Limits for Payload Length and Timeout . . . 86
6.7 FEC 88
6.7.1 Forward Error Correction (FEC) Encoding 89
6.7.2 Timing Receiver (TR) 90
6.8 Etherbone Slave and Event Condition-Action Unit 91
6.9 White Rabbit Network Model 91
6.9.1 Interference at Network Integrated Controller (NIC) 91
6.9.2 White Rabbit (WR) Switches 93
6.10 End-to-End Delay Analysis 93
6.10.1 Earliest Deadline First (EDF) Sink Tree 94

DRAFT

6.10.2 Equivalent Circuit for WR Network 97
6.10.3 Data Master to Timing Endpoint 98
6.10.4 Equivalent Service of Packetised Greedy Shapers . . 104
6.10.5 Aggregate Scheduling 104
6.10.6 Summary 106

7 Memory Maps **107**

8 Quick Reference **116**

9 Troubleshooting **121**

I Attached Documents **122**

II Document Information **123**
II.1 Document History 123

DRAFT

Chapter 1

Introduction

The new Facility for Antiproton and Ion Research (FAIR) facilities required a new Control System (CS), which is currently implemented and tested as an upgrade to the facilities of the GSI-Helmholtz Centre for Heavy Ion Research (GSI). This document described the real-time CS, specifically the General Machine Timing (GMT), as implemented and used in the 2019 beamtime. The primary focus lies on the Data Master (DM), which generates and distributes commands to be executed on the time-synchronised Frontend Controller (FEC)s.

This document is both a user manual for the FAIR DM and a collection of research and implementation documents. The latter are providing a deeper understanding of the DM's concepts, functions and algorithms. It also provides material for a hands on introduction to the tools and API. Chapter 2 is therefor written as a tutorial, meant as a quick start for anyone who needs to adminstrate or debug a DM. The appendices contain additional information, tables and memory layout documentation complementing the doxygen documentation of the carpeDM source code.

1.1 General Machine Timing

The GMT encopmpasses all real-time (RT) aspects of controlling accelerator hardware. The concept of RT itself is often misinterpreted, as the term only means a requirement for determinism. A reaction has to occur within a defined timeframe of a stimulus, but it does not tell anything about how long this may take. In this context, RT will always be understood as "hard real-time". "Hard" means that data which has not been processed before

its deadline is due, has no value anymore and is even considered dangerous. As an example, an autonomous car is a hard RT system. A distance value from a sensor has little to no value it became available late, because the car's perception would be lagging behind physical reality. Likewise, the command to activate the breaks being late would make it useless. It would no longer fulfill the purpose of avoiding a collision.

Deadline windows in the FAIR-CS vary depending on the device and assigned tasks, but as a rule of thumb, RT units are expected to react somewhere in between the low millisecond to low microsecond range, with an accuracy in the nanosecond range. The RT section of the FAIR-CS is currently undergoing an upgrade from older, MIL-STD-1553 (MIL)-bus based technology to the new White Rabbit (WR) system. With it comes a change in paradigm, from event based to alarm based machine control.

1.1.1 New vs Old Design Philosophy

The underlying concept of the new GSI/FAIR CS was to create an alarm based CS, as opposed to the previous event based system. The new architecture aims for accurate hard RT control of machines, not influenced by the machine type, distance, controller form factor or interface type.

Old – event based In an event based system such as in figure 1.1 the transmission time of an event is part of the control loop. An event is sent and the receiving system will carry out an action upon arrival. For multiple synchronous actions, multiple events must arrive synchronously. This is usually achieved by matching signal propagation delays, either by matching cable lengths or introducing delays on the faster routes. The advantage lies in simplicity and low overhead during runtime; a form time synchronisation is not necessary. The downside lies in the network topology (i.e., its delays) being part of the event handling. This make configuration and expansion of the control network difficult, very hard to automate and time consuming.

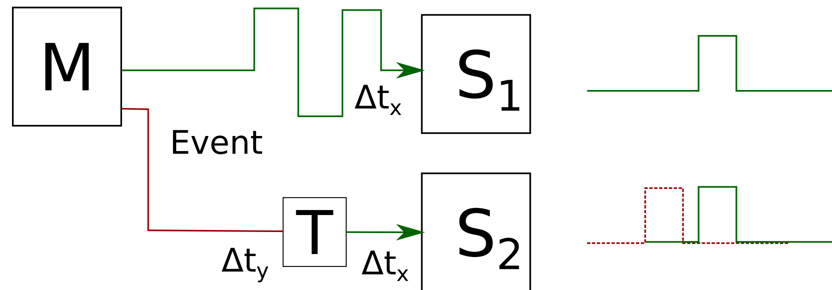


Figure 1.1: Event based System, Master (M) sends events to multiple slaves(S). Lines with lower delay (red) need to be compensated

New – alarm based In an alarm based system as shown in figure 1.2 on the other hand, all messages contain an absolute deadline (alarm), describing when they are to cause an action in the receiver. Upon reception, messages are stored until their alarm is due and their action is executed. For multiple synchronous actions, the lead time for message dispatch can be roughly chosen and just needs to be greater than the maximum possible transmission delay in the network. The great advantage is the ease of configuration due to the independence from transmission delays (temperature, network traffic, change of topology). This allows very high scalability, easy administration and very accurately synchronised actions.

The downside of an alarm based system lies in the necessary complexity of senders and receivers. Clock oscillators and absolute time must be accurately synchronised between nodes and alarm messages require more complex protocol handling than events. This also results in a higher price per receiver unit than an event based approach.

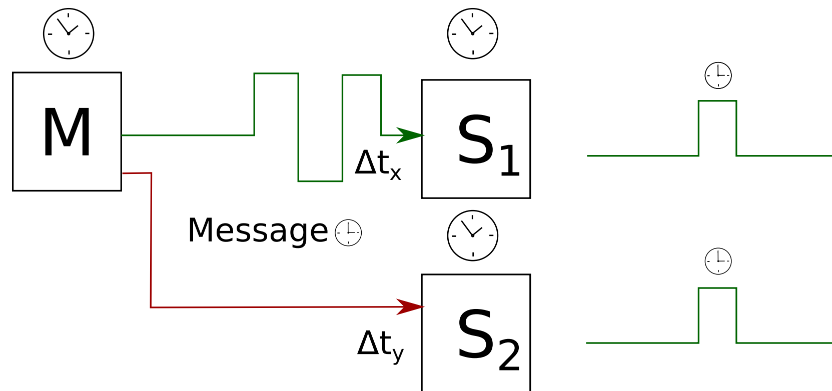


Figure 1.2: Alarm based system. Master (M) sends messages to multiple slaves(S) in advance. Line delay has no effect on timed message, node time must be synchronised.

1.1.2 Responsibilities

The new CS design splits responsibilities for machine control into three distinct systems (see 1.3).

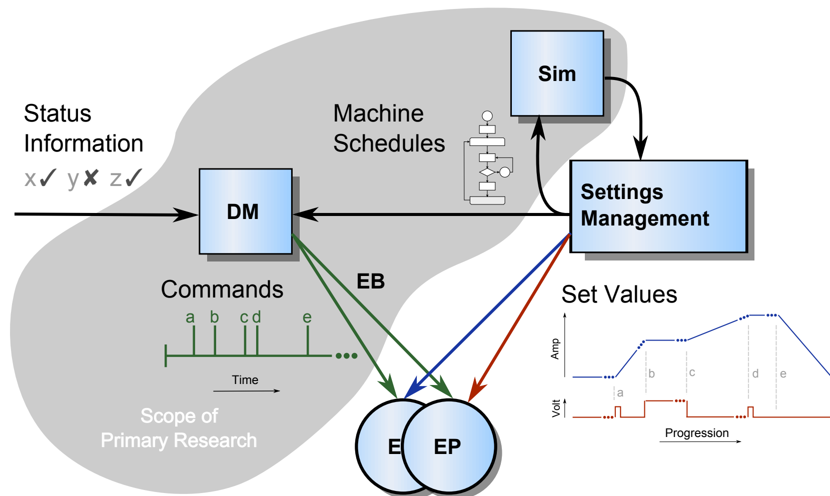


Figure 1.3: GSI/FAIR CS Design

Settings Management delivers sets of configuration values and curves to machines (frequency and phase settings for Radio Frequency (RF), current ramps for magnets, etc). This requires large amounts of data and thus high bandwidth traffic, but the timeframe for delivery is relaxed.

Timing uses a separate, deterministic network to synchronise the local time and clock oscillators of master units and all endpoints on site. It allows a heterogenous Timing Receiver (TR) pool as well as arbitrary geographic and network topologies.

Command uses the same network as timing and delivers command messages ahead of time to endpoints. Upon arrival, their alarm is queued and dedicated hardware modules guarantee action execution to 1 ns accuracy. Command can either directly control output ports or select machine data sets and order local firmware to handle execution.

1.2 Timing System and the Datamaster

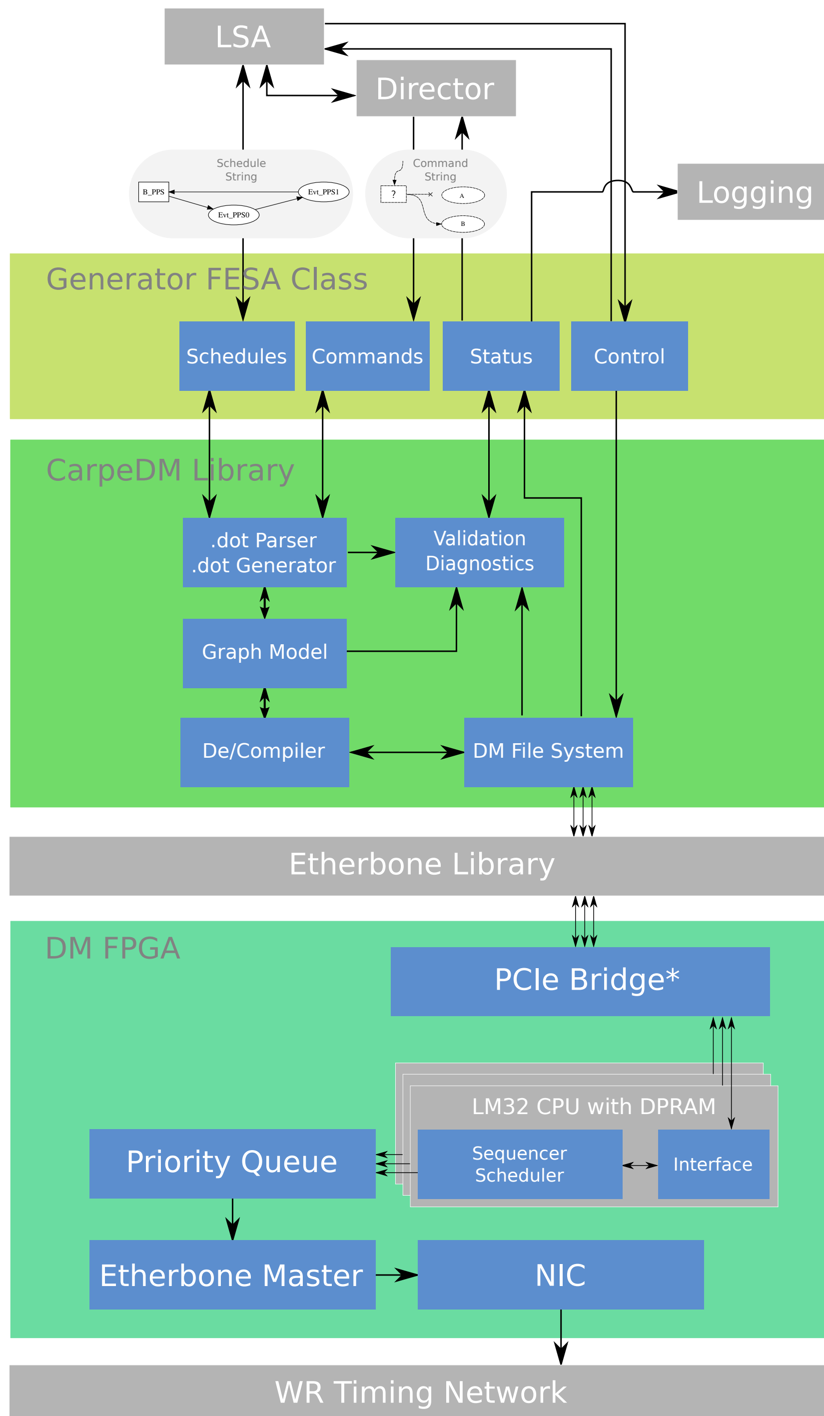


Figure 1.4: Schematic of the DM components

1.2.1 Control System Stack

The stack between Large Hadron Collider (LHC) Software Architecure (LSA) and the timing network consists of a multitude of layers. The most important ones are the schedule parser, the graph model, syntactic and structural analysis, offline timing analysis, runtime control and validation, de/serialisation, hardware (HW) processing and network streaming. Figure 1.4 shows the individual layers of the DM, going from abstract to real-time from top to bottom. The high level connections to the LSA and Director black boxes provide the input. The Generator Frontend Software Architecture (FESA) class, CarpeDM Library and Etherbone (EB) library all run on the DM server (x86_64), which runs a standard frontend Linux without real-time extensions. Layers below the host system run in programmable hardware and are real-time WR capable. By default, the used TR is PEXARIA V Peripheral Component Interconnect express (PCIe) board. Access to the board's System on a Chip (SoC) is available via PCIe. carpeDM can also be connected to remote DM hardware instead. The connection can be run over WR network, over TCP, using the host platform as a software (SW) bridge (via socat) to PCIe. The Generator FESA class and the EB library will only briefly be described in this documentation. EB is well documented already **123**, while the Generator FESA class was intentionally designed as a "stupid" middleware wrapper for carpeDM. Apart from some additional logger and formatting code, all functionality is borrowed from the carpeDM library.

1.2.2 Building Blocks

High level LSA is in essence a physics modelling framework for accelerators. A simplistic explanation is that models of accelerator components (physical properties of magnets, RF cavities, power supplies...) and the desired beam properties are entered on one side, the necessary machine settings and command sequences come out on the other. The resulting actions are supposed to run in parallel, as alternative scenarios, or linked by a form of handshake. As described in 1.1.1, this needs both settings data, which is usually present in form of curves and tables, and a command sequence at runtime, choosing which settings data set is to be used where and how. LSA's output to the timing system are "schedules". These are programs in a domain specific language, describing timing command generation, whose execution wihtin the DM results in a stream of commands to timing receivers. Schedules are represented as graphs in carpeDM. To make it a good match to LSA content, this graph representation needed

to be abstract, flexible and powerful. To make the DM's command distriubution deterministic however, the data structures used on the lower (embedded system) level needed to be simple and efficient. Lossless bidirectional translation between high level and low level representation was also a requirement.

Low level The DM HW is the embedded system in charge of creating the stream. There is a strong separation between the actual real-time sequencer, which must never be disturbed from outside (no blocking calls) and the command interface. The DM consists of multiple embedded Central Processing Unit (CPU)s, each with their own independent Random Access Memory (RAM). To guarantee that the host does not block the DM HW, each RAM features two completely independent physical ports, removing bus access as a bottleneck. Using techniques borrowed from inter-thread communication models, the command module interacts with the realtime sequencer by message boxes and flags inside the Dual Port RAM (DPRAM).

1.3 Language for Accelerator Control

carpeDM was designed as a domain specific language, using directed graphs to represent machine schedules at a high level. The language is not Turing-complete, but has distinguishing features very well suited for real-time control. carpeDM provides timely command generation and dispatch, conditional branches, nested loops, inter-process communication and real-time synchronisation. It was designed for extensive parallelism. carpeDM also allows thread and transaction safe manipulation and replacement of partial schedules (subgraphs) at runtime. Future upgrades will include a full a-priori worst case analysis of processors loads, bus and network traffic to ensure proper functionality even before machine schedules are executed.

1.3.1 Schedule Graphs

Schedule graphs are at the core of carpeDM, and this representation of a control program comes with two advantages. The first is that graph algorithms are a well researched field, which provides a rich set of tools to construct, search, manipulate and verify graphs. The second advantage is right in the name – graphs are meant to be visualised, and visualisation helps understanding.

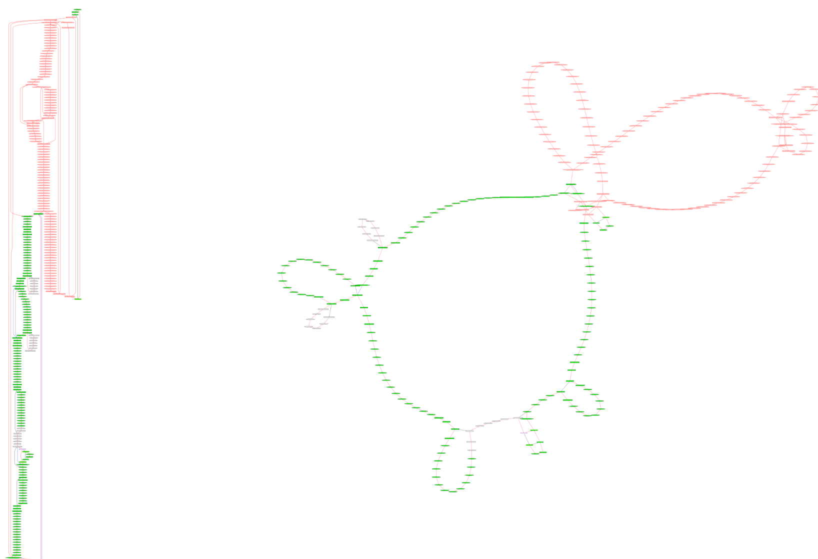


Figure 1.5: The same schedule graph shown in different visualiser settings. On the left: *dot*, right: *neato*.

Benefits of inbuilt visualisation Using the graphviz library and tools, there is a wide range of visualiser types and styles available. Figure 1.5 shows the same graph visualised with two different sample settings. The *dot* visualiser on the left with its flow-chart like representations helps showing flow directions and dependencies and is well suited for debugging flows within a single control program. Others, such as *neato* on the right, provide a more organic style, reminiscent of organic structures and best suited for large graphs. This style is best suited for seeing the extent of wait loops, alternative scenarios and the interplay of multiple control programs and machines.

1.3.2 Look, feel and function

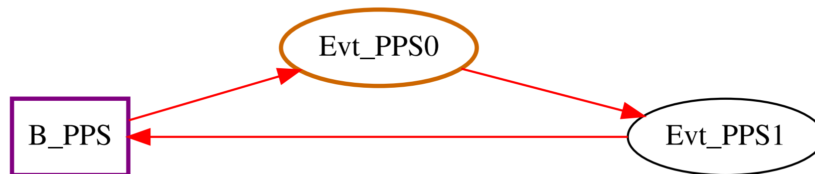


Figure 1.6: Visualisation of Hello World Schedule Graph

Shape and coloration are effective tools to further understanding. Let's have a look at the hello world example in figure 1.6. There are two different node shapes there, rectangles and ovals. Ovals represent timing messages, rectangles are blocks, representing timespans and serve as decision points. The nodes are connected by directed edges (arrows), the edge colour represents relations between nodes. For example, a red arrow denotes an active path, black marks an inactive alternative path, blue leads to a communication target and so on. Likewise, node fill or frame colours are used as a visual aid. For example, a green fill is used to indicate that DM embedded system has processed a node at least once. This painting of territory is often a great help in coverage tests and understanding the taken course through a schedule. A full legend of nodes, edge types and their appearance can be found at 2.3.

Another good example is the safe2remove module of carpeDM (see 4). It analyses the schedule runtime and determines if a given subgraph can safely be manipulated or removed. This is achieved by converting all activity in time into a time-invariant equivalency graph. The underlying verification algorithm has to convert run time uncertainties into a static worst-

case image, which is a non trivial task. Its debug outputs are therefore complex. To just name a few examples, the analysis of the hierarchy of dynamic commands, found predominant paths, transformation log of inbuilt and runtime commands, extrapolated future safe states, contracts with the user regarding which queued commands must be preserved, safety assessment base on union and difference sets . . . the list of complex (even to developers) activity logs goes on. When a new feature is implemented, failed test cases need to be understood to correct errors, and doing this from a vast number of debug messages is difficult and time consuming.

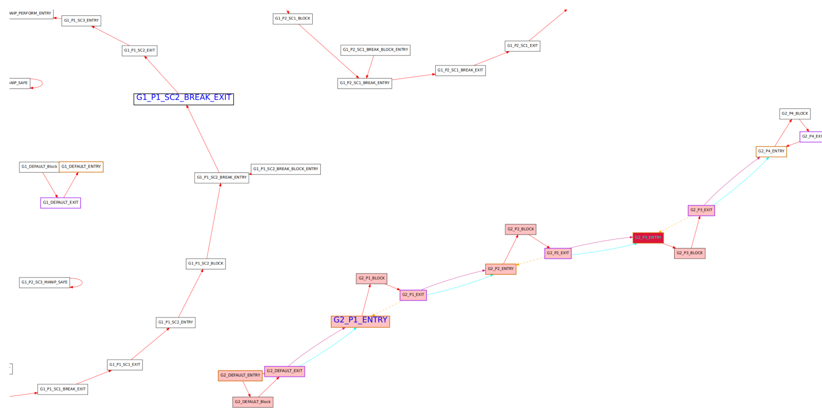


Figure 1.7: Visual style for validation of runtime subgraph changes

Visualised, however, the systems's actions and decision quickly become tangible: All dynamic activity is converted to edges of specific colours and pen styles. All nodes reachable from the subgraph via traversible edges are marked in red – *dangerous territory*. The execution cursors are marked in blue – *important*. If no blue markers are inside in red territory, manipulating the subgraph has no negative impact and is allowed. If one or more are inside, manipulation would endanger the DM embedded system and is forbidden. Simple.

1.3.3 Graph translation for an embedded system

carpeDM uses a simplified file system in the embedded level, inside which each graph node occupies (very small at 52 B) page in memory, avoiding fragmentation while reaching near optimal memory utilisation. Such small sizes might seem odd, but one must keep in mind that RAM inside an Field Programmable Gate Array (FPGA) is not like RAM inside a normal computer. It can be used highly compartmentalized and can be

accessed truly parallel on a HW level, but its size is extremely limited. The current DM is limited to 4 MB.

The graph structure is converted into binary structures. Directed graphs can easily be turned into linked lists, which work well in combination with the lean minimal file system. All intelligence (and space) needed for memory management and transaction management is placed at the host side. The embedded sequencer can therefore follow a very simple design. It consists solely of a scheduler and several worker threads per CPU which follow pointers through linked lists of memory chunks representing the original graph. Instead of a stack, the sequencer uses local storage queues for all dynamic change requests, one at each point of decision in the graph. If it is be guaranteed that the communication queues are cleaned up after decisions are revoked at runtime, this distribution of control has several advantages. In particular, it allows fine control of subgraphs, which can be individually added or removed during runtime. It is further possible to link otherwise independent operations with a handshake, such as beam acceleration in the Schwerionen Synchrotron 18 (SIS18) synchrotron rings and extraction for storage in the Experimentier-Speicherring (ESR). The current implementation also features a fast EB runtime interface for other time critical devices in the control system, such as the UNILAC gateway and Bunch-to-Bucket (BTB) control, or later, machine protection systems.

Chapter 2

CarpeDM User Guide

carpeDM is a framework for a domain specific programming language, designed to interface with both the DM and LSA. It's purpose is to provide a description format for accelerator schedules, manipulate the resulting graphs and compile/decompile them for use in the DM. carpeDM validates syntax, grammar and structure for incoming graphs. It also handles runtime commands to the DM and assures transaction safe manipulations. It also handles DM memory management and content loading them them back and forth between the LSA physics model and the real-time hardware (HW) of the DM. carpeDM also handles data transmission to and from the DM and manages the DM's HW resources (memory, bandwidth, etc.). The name carpeDM was also used for the corresponding c++ library.

Representation carpeDM uses *dot* graphs as defined by the graphviz organisation. Dot graphs are a generic format for directed or undirected graphs. Each graph consists of nodes and connecting edges, as well as style and layout parameters. Style parameters are automatically generated into carpeDM's output files for ease of understanding, but are ignored on input files.

An opensource suite of tools to generate graphic representations of dot files is freely available. For human interaction (operators) and especially in the deployment phase, these visualisations already were of great value to reduce the time to determine the facilities state and debug its behaviour.

2.1 Getting Started

2.1.1 Installing carpeDM Tools

First, you will need the BEL projects sources from the GSI controls repository. If you have not already done so, you can get the sources here:

```
$ git clone --recursive
   https://github.com/GSI-CS-CO/bel_projects.git
$ cd bel_projects
$ git checkout master
$ ./fixgit
```

To build and run carpeDM, you will need the boost libraries $\geq$ 1.5.4 (if you build on the ASL cluster, these are already installed).

```
$ sudo apt-get install libboost
```

Next, you'll need to build the toolchain and the carpeDM library and tools. From the root folder of your BEL projects checkout, call

```
$ make
$ cd syn/gsi_pexarria5/ftm
$ make tools #To also get DM gateware, run plain make instead
$ sudo make install
```

This leaves you with the carpeDM library *libcarpedm.so* and two command line tools, *dm-sched* and *dm-cmd*.

2.1.2 Installing Visualisation

There's two options how to use dot visualisation. If you are interested in viewing, navigating and searching through graphs, you'll want the xdot viewer. This python app is lightweight and easy to use. If you need renderings of a graph for documentation or want more control over all the render parameters, you should use the graphviz tools directly on the command line.

Installing all prerequisites First, we need to install the graphviz package, which comes with several CLI render programs (dot, neato, fdp, twopi, circo, sfdp)

```
$ sudo apt-get install graphviz
```

It is sensible to render dot files into a vector format (pdf, svg, etc), as bitmaps of schedules tend to get *very* big. The graphviz CLI tools accept parameters for the renderer, which are grouped into graph, node and edge

parameters. A lot of the parameters are specific to one renderer, and are ignored if you run another. These can also be supplied in the graph itself, however, CLI parameters always override. For example, *-Grankdir=TB* will cause the dot renderer to arrange the graph top to bottom instead of left to right. Especially the neato renderer is very useful for the large sequences found in schedules, but needs some extra parameters to produce sensible results.

```
$ neato <a-dot-file.dot> -Goverlap=compress -Gmodel=subset
```

You can try adding several moree parameters to the call to refine node distance, edge spring force, different arrangement models and so on

```
$ ./dotrender.sh download &
```

To get a live view of the DM's content, open download.svg in a viewer supporting auto refresh. If you have larger and more complex schedules, you might want a different layout to get a better overview. In this case, try the *neatorender.sh* script instead. It will produce a more "organic", more compact graph that is better suited to get the big picture.

The Xdot Viewer The second option is to use the Xdot Viewer tool. This is a Gtk 3 based GUI viewer written in python, providing a much more comfortable interface to schedule graphs. It is also faster than using the CLI renderer plus an image viewer. For carpeDM, a fork has been made of the original xdot project, adding new features.

```
$ sudo apt-get install python3
$ sudo apt-get install python3-setuptools
$ git clone https://github.com/GSI-CS-CO-Forks/xdot.py.git
$ cd xdot
$ git checkout xdot-gsi-dm
```

```
#Try it out
$ python3 -m xdot <a-dot-file.dot>

#Install
$ sudo python3 setup.py install --record files.txt

#Uninstall
$ cat files.txt | xargs sudo rm -rf
```

2.1.3 Xdot Viewer Short Manual

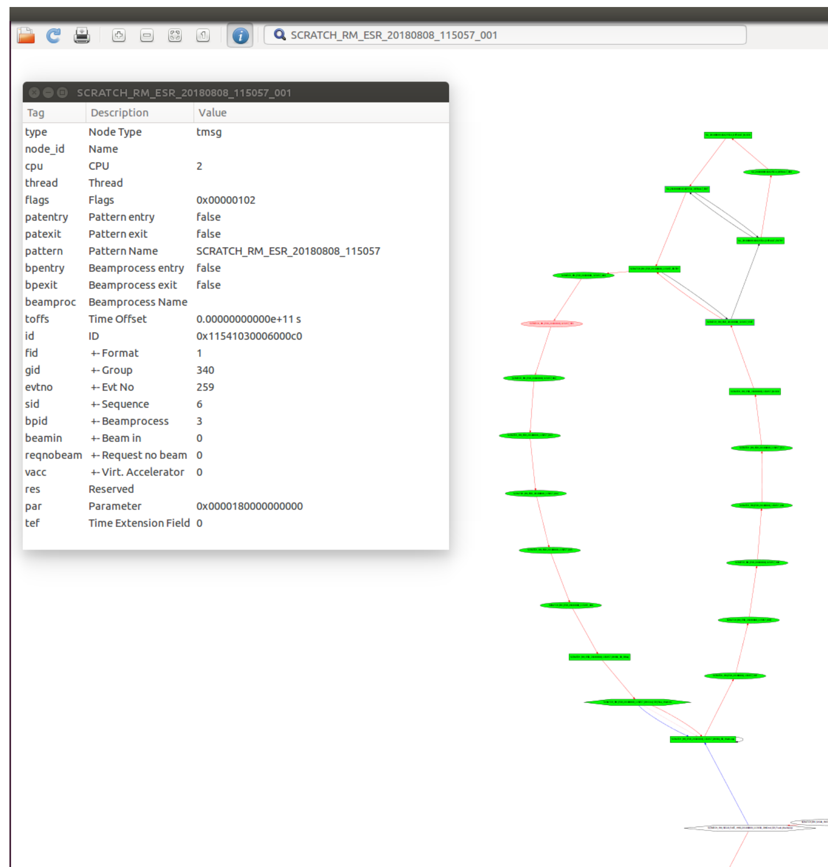


Figure 2.1: Xdot Viewer

From the installation on, you can call xdot like any other program. There are CLI option to choose the renderer and pass arguments to it. The following call will start xdot with a neato configured for schedule graphs:

```
$ xdot <a-dot-file.dot> -f neato
    --filterargs="-Goverlap=compress -Gmodel=subset"
```

Files and Printing The File dialogue can be used to load dot files and is pretty much self explanatory. Printing is still buggy, as it only chooses the correct print area if the window it at its original (when opening the program) size. Future releases should feature an SVG export

Navigation The arrow keys can be used for scrolling, as can the pressed middle mouse button. Zooming is achieved by using <PageUp>/<PageDown>

or using the mouse wheel. In addition, the toolbar buttons can be used to control zoom.

Copy Name A right click on a node will copy its name to the clipboard.

Inspection Window Clicking the info button in the toolbar will open the IW. Left clicking a node or edge will show its bundled properties. The window contains three columns, description, tag and value. Descriptions are only available for carpeDM properties and explanatory comments. The tag column is the actual tag in the dot file, the value is the value from the dotfile. Note that the value can be converted. For example, time is not shown as a integer of nanoseconds $1\,003\,000\,\mathrm{ns}$, but as seconds in scientific notation $1.003 \times 10^{-6}\,\mathrm{s}$.

Text Search The toolbar provides a text search field to search for nodes and edges. <Enter> begins the search and will zoom/scroll in the group of found nodes or edges. All found items are highlighted in light red. Text search also allows the use of regular expressions. For example,

```
SIS18_RING_.*00.
```

will search for all node names starting with *SIS18_RING* and ending in *00x*.

When the inspection window is active, text search will also search bundled properties. They are stored internally as strings of the format <key=value> and can be searched as such. Regular expressions are very powerful when used as boolean connections of search criteria. For example,

```
gid=300|gid=508
```

will search for all nodes belonging to group ID *300* or *508*, while

```
(?=.*gid=300)(?=.*evtno=258)
```

will find nodes of group *300* and having an event number of *258*.

2.1.4 Hello World!

We will assume that you have a freshly booted (or halted and cleared) DM available over an EB connection. If you encounter any error messages, especially during status check, please look at appendix 9, Troubleshooting.

First encounter First, let's have a look what the DM thinks it's doing right now The following command will give you the detailed runtime status report.

```
$ dm-cmd <eb-device> status -v
```

The output will look something similar to the listing 2.1. Note that none of the worker threads is running right now and none has assigned a pattern or node to it.

```
╔══════════════════════════════════════════════════════════════════╗
║ DataMaster: dev/ttyUSB0            │ WR-Time: Tue Feb 6 17:50:44 ║
╠══════════════════════════════════════════════════════════════════╣
║ Cpu │ Thr │ Running │ MsgCount │    Pattern    │            Node ║
╠══════════════════════════════════════════════════════════════════╣
│  0  │  0  │   no    │        0 │  undefined    │            idle │
│  0  │  1  │   no    │        0 │  undefined    │            idle │
...
│  3  │  7  │   no    │        0 │  undefined    │            idle │
└──────────────────────────────────────────────────────────────────┘
```

Listing 2.1: Output of dm-cmd status

```
$ dm-sched <eb-device> add helloworld.dot
```

carpeDM will parse and validate the dotfile, create the graph and upload it to the DM. It then reads back the binary, transforms it into graph again, annotes it with visualisation tags and writes it to download.dot. The result should look similar to figure 2.2.

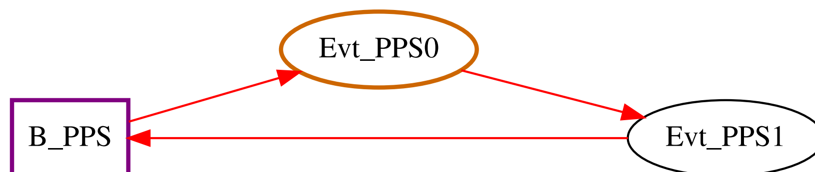


Figure 2.2: Visualisation of Hello World Schedule Graph

The DM yet has to be told that we wish to play the Hello World Pattern. Dots are also used to describe runtime commands to the DM, and we shall use prefabricated ones in this example.

```
$ dm-cmd <eb-device> -i helloworld_start.dot
```

This started the pattern execution. Let's check the status again, this time without the verbose flag:

```
$ dm-cmd <eb-device> status
```

The DM is now sending two messages once every second, with an execution time 8 ns apart. If the output is connected to a TR over a WR switch, such as an SCU, you can log into your SCU and see the events caused by our messages coming in. For this to happen, you need to tell the *saft-ctl* tool what you wish to see. In our case, we filter to show only our own hello world events.

```
$ ssh root@<Your SCU's Name>.acc.gsi.de
SCU$ saft-ctl tr0 -v -f snoop 0x0 0x0 0x0
```

```
tDeadline: 2018-02-06 17:08:10.556617664 ... EVTNO:  280 ...
tDeadline: 2018-02-06 17:08:10.556667664 ... EVTNO:  273 ...
    !delayed (by 565680 ns)
```

Yep, there they are! Note that the second message has a comment saying it is *delayed*. This is nothing to worry about. The the snooping action by the SCU is way slower than the TR hardware, unable to process another message only 8 ns after the first. All status reports by TR's are explained in detail in chapter **??**.

Congratulations, you just ran your very first accelerator schedule with carpeDM and saw the result on real hardware!

2.2 Command line tools

carpeDM comes with two command line tools, *dm-sched* and *dm-cmd*. dm-sched is responsible for schedule upload, download and manipulation. dm-cmd covers manual thread and flow control, status queries, runtime diagnostics and node and queue inspection. For detailed help, call either with the "-h" flag.

to be continued...

2.3 Schedules

2.3.1 Overview

carpeDM schedules use nodes of three basic types to model the control message stream to the accelerator. These are *Messages*, *Commands* and *Blocks*. All necessary management overhead is contained in nodes of a fourth *Meta* type, which is by default invisible to the user.

Message Nodes Messages, aka timing messages, provide what it says on the box - they create messages to be broadcasted to timing receivers on the White Rabbit (WR) network.

Block Nodes Blocks provide three functions in one. First, they carry a timespan or period, which is added to the running time sum once they are processed. Second, they can be outfitted with a sink for commands, allowing dynamic actions. These could be a request to wait or changes to the flow through the graph (alternative successor nodes). And third, Blocks can be made to dynamically adjust their period to fit a given time grid.

Command Nodes Command nodes use the same command interface as Blocks do, but they are sources, not sinks. They can be used for synchronisation or loops with various properties. An example would be a loop waiting for an external command to continue, but terminating when reaching a given timeout.

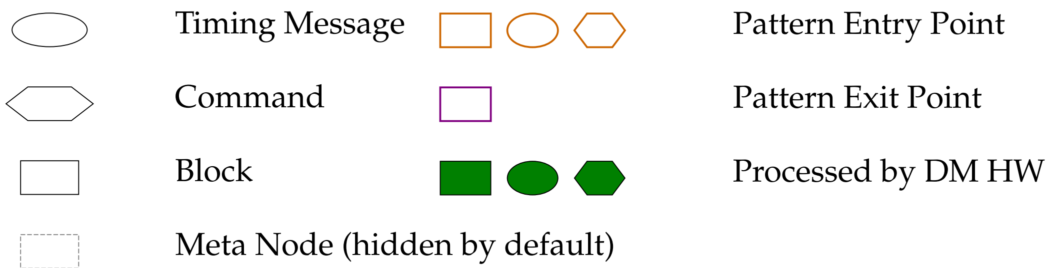


Figure 2.3: Legend for Node Visualisation

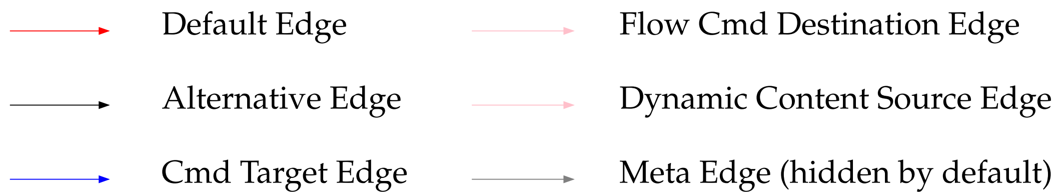


Figure 2.4: Legend for Edge Visualisation

2.3.2 Basics

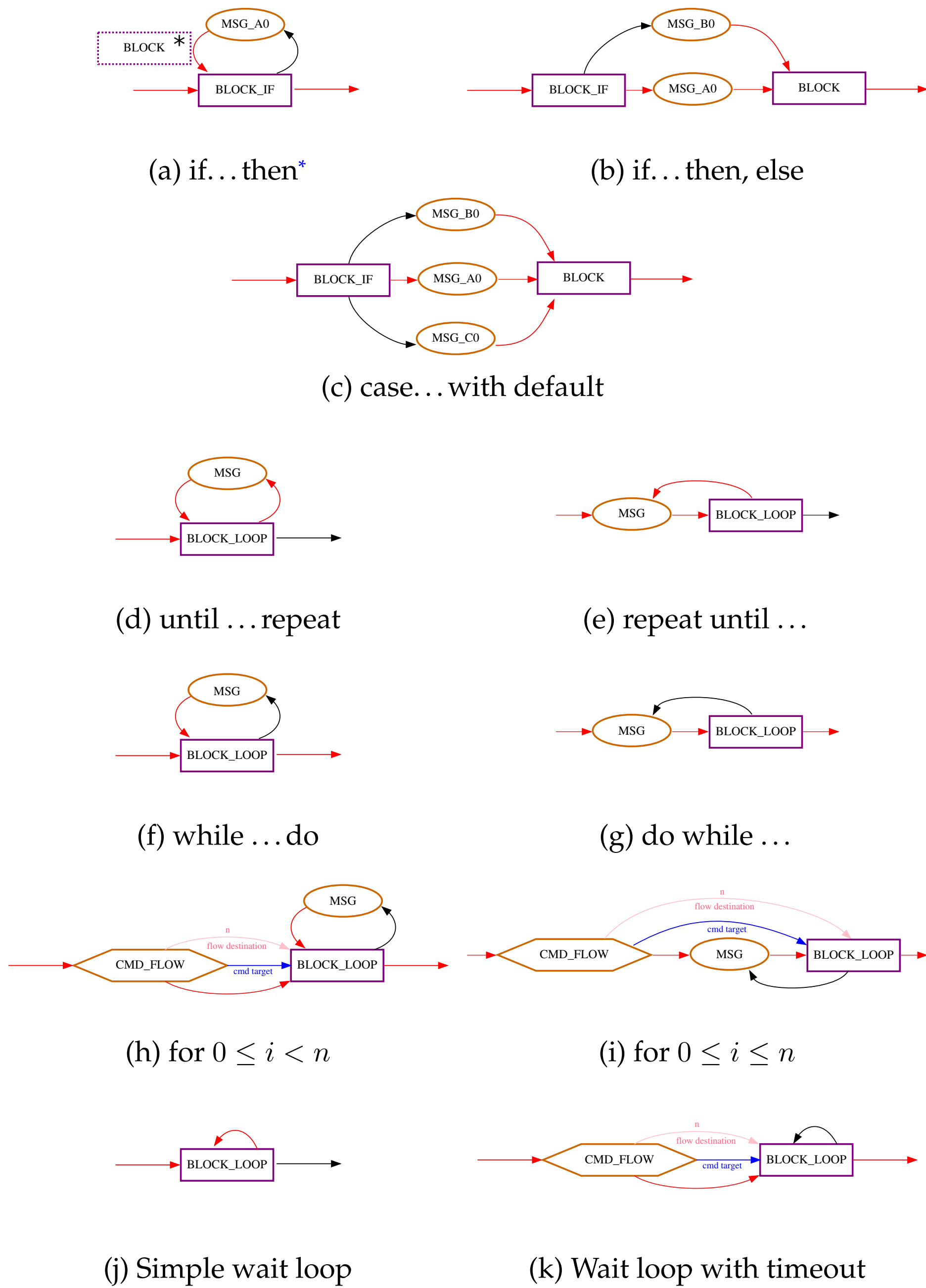
2.4 Flow Control

Like other program languanges, carpeDM schedules support branches and loops. There is no generic conditional conditional check when deciding wether to take a branch though. Instead, a message inbox, the command queue, is checked for new orders. This means that commands are queued at the point in the graph where the change is to be made. To allow parallel operation, there are as many command queues as there are points of decision. Points of decision are always of the "block" type, but blocks are not always points of decision.

2.4.1 Blocks and Changes during Runtime

For blocks to be used dynamically, they need to act as a sink for commands. This is enabled by adding command queues to the block. Up to three priorities are supported, forming a single priority queue. When a block is processed, it will only ever execute a single command and will always choose the highest priority pending.

Commands themselves are in fact command generators, each represents $0 \ldots n$ repetions of a command. Although only certain command type can be (sensibly) executed multiple times, all commands share the generator trait. This means they are functions which have an internal state (their repetition counter, aka quantity). Such a command generator will yield the same command every time it is executed until its quantity reaches zero. Generators with a quantity of zero are exhausted and popped from the queue. This behavior is required for the use of commands as loop initialisers, see **??**.



(a) if…then*

(b) if…then, else

(c) case…with default

(d) until …repeat

(e) repeat until …

(f) while …do

(g) do while …

(h) for $0 \leq i < n$

(i) for $0 \leq i \leq n$

(j) Simple wait loop

(k) Wait loop with timeout

Figure 2.5: **Schedule cheatsheet**

---

*Add optional blocks to achieve different durations of alternate paths

2.4.2 Branches

TODO complete missing examples!

Using the *flow* command, blocks can temporarily or permanently change their successor node. Figure 2.6 (exbranch.dot) shows a minimal example containing two alternative branches. The default branch taken contains the 'A' nodes, and block BLOCK_BRANCH features a single low level command queue (meta nodes shown for demonstration). Changing the flow from the 'A' to the 'B' branch can be achieved by sending a command to block BLOCK_BRANCH.

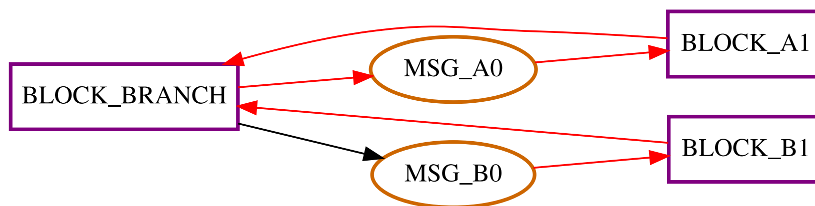


Figure 2.6: Example of simple branch

```
$ dm-sched <eb-device> add exbranch.dot
$ dm-cmd <eb-device> startpattern BRANCH
$ dm-sched <eb-device>
```

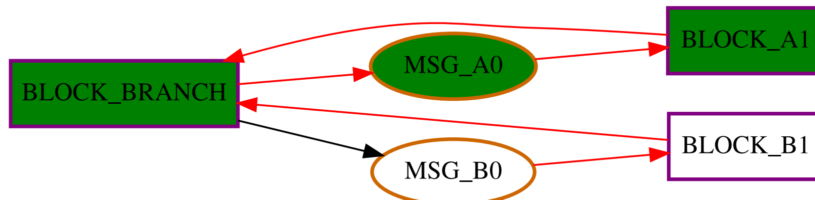


Figure 2.7: Default flow through 'A' branch

```
1  digraph g {
2  name="BranchExample";
3  graph []
4  edge  [type="defdst"]
5  node  [cpu="0"];
6
7    MSG_A0      [type="tmsg",  pattern="A", patentry="true", toffs  = 0, fid=1, gid
       =4048, evtno=1, par="0"];
8    BLOCK_A1    [type="block", pattern="A", patexit="true",  tperiod=100000000];
9    MSG_B0      [type="tmsg",  pattern="B", patentry="true", toffs  = 0, fid=1, gid
       =4048, evtno=2,  par="0"];
10   BLOCK_B1    [type="block", pattern="B", patexit="true",  tperiod=100000000];
11   BLOCK_BRANCH [type="block", pattern="BRANCH", patentry="true", patexit="true",
       tperiod= 20000000, qlo="1", qhi="1", qil="1"];
```

```
12
13    BLOCK_BRANCH -> MSG_A0;
14    MSG_A0 -> BLOCK_A1 -> BLOCK_BRANCH;
15    MSG_B0 -> BLOCK_B1 -> BLOCK_BRANCH;
16    BLOCK_BRANCH -> MSG_B0 [type="altdst"];
17  }
```

Listing 2.2: Branch

Calling dm-sched on a DM without any further parameters will automatically call the status report, which will make the render script update the graph image. The green fill in figure 2.7 shows that the DM followed the red default edges as expected and executed the 'A' branch at least once, but did not enter the 'B' branch. We can now change the flow by

```
$ dm-cmd <eb-device> flow BLOCK_BRANCH MSG_B0
#or
$ dm-cmd <eb-device> flowpattern BRANCH B
#followed by
$ dm-sched <eb-device>
```

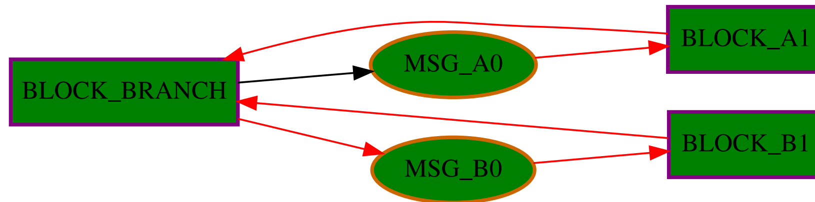


Figure 2.8: Flow changed to 'B' branch

We can see now that the DM changed the default path towards the 'B' branch, the green fill showing it was executed.

2.4.3 Loops

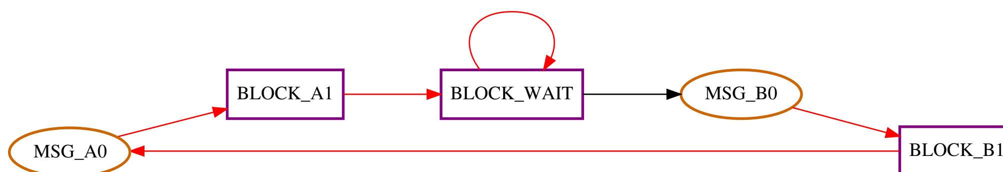


Figure 2.9: Wait Loop

```
digraph g {
name="WaitLoopExample";
graph []
edge  [type="defdst"]
node  [cpu="0"];
  MSG_A0      [type="tmsg",  pattern="A", patentry="true", toffs  = 0, fid=1, gid
    =4048, evtno=1, par="0"];
  BLOCK_A1    [type="block", pattern="A", patexit="true",  tperiod=100000000];
  MSG_B0      [type="tmsg",  pattern="B", patentry="true", toffs  = 0, fid=1, gid
    =4048, evtno=2, par="0"];
  BLOCK_B1    [type="block", pattern="B", patexit="true",  tperiod=100000000];
  BLOCK_WAIT [type="block", pattern="WAIT", patentry="true", patexit="true",
    tperiod= 20000000, qlo="1"];

  MSG_A0 -> BLOCK_A1 -> BLOCK_WAIT;
  MSG_B0 -> BLOCK_B1 -> MSG_A0;
  BLOCK_WAIT -> BLOCK_WAIT;
  BLOCK_WAIT -> MSG_B0 [type="altdst"];
}
```

Listing 2.3: Wait Loop

The most basic example of command-controlled loop is an infinite loop. It is executed until an incoming flow command orders the DM to leave the loop. Figure 2.9 shows the setup. A Block with its default edge pointing at itself is forming an infinite loop. Note that only blocks are allowed to have themselves as a successor. The loop can be left by sending the block a flow command, which will order the DM to the node Msg_CONTINUE. The flow in the example is temporary, it does not change the default destination. This allows the wait loop to be used again without further action required. The period of wait loops must be chosen greater than the maximum time the DM's scheduler requires to process the block. A value of 10 µs or more is recommended.

```
$ dm-sched <eb-device> add exwloop.dot
$ dm-cmd <eb-device> startpattern LOOP
$ dm-sched <eb-device>
```

2.4.4 Default Pattern Example

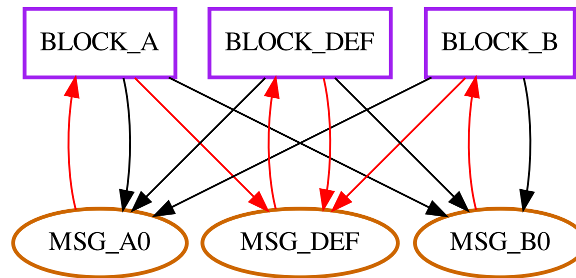


Figure 2.10: Default pattern with two alternatives

```
digraph g {
name="DefPatExample";
graph []
edge  [type="defdst"]
node  [cpu="0"];

  MSG_DEF   [type="tmsg",  pattern="DEF", patentry="true", toffs  = 0, fid=1, gid
    =4048, evtno=0, par="0"];
  BLOCK_DEF [type="block", pattern="DEF", patexit="true", tperiod= 20000000, qlo=
    "1"];
  MSG_A0    [type="tmsg",  pattern="A", patentry="true", toffs  = 0, fid=1, gid
    =4048, evtno=1, par="0"];
  BLOCK_A  [type="block", pattern="A", patexit="true",  tperiod=100000000, qlo="1
    "];
  MSG_B0    [type="tmsg",  pattern="B", patentry="true", toffs  = 0, fid=1, gid
    =4048, evtno=2, par="0"];
  BLOCK_B  [type="block", pattern="B", patexit="true",  tperiod=100000000, qlo="1
    "];

  MSG_DEF -> BLOCK_DEF -> MSG_DEF;
  BLOCK_DEF -> MSG_A0 [type="altdst"];
  BLOCK_DEF -> MSG_B0 [type="altdst"];
  MSG_A0 -> BLOCK_A -> MSG_DEF;
  BLOCK_A -> MSG_A0 [type="altdst"];
  BLOCK_A -> MSG_B0 [type="altdst"];
  MSG_B0 -> BLOCK_B -> MSG_DEF;
  BLOCK_B -> MSG_A0 [type="altdst"];
  BLOCK_B -> MSG_B0 [type="altdst"];

}
```

Listing 2.4: Default pattern with two alternatives

Based on the examples of branches and simple wait loops, we can construct a scenario using a default pattern and alternative patterns which

will only be played on request. The basic principle is the same as the wait loop, with the difference of the loop being a productive sequence. This is a common case for FAIR, where a default pattern is played whenever no beam requests are demanding other patterns (d-d-d-d-A-d-d-A-B . . .).

You might have noted that we did not start the DM this time, but still issued the flow command. It will lie in wait inside the default patterns command queue until the block is evaluated. The command line tools also provide a way to take a peek at the queue content. Using the following command

```
$ dm-cmd <eb-device> queue BLOCK_def
```

```
Inspecting Queues of Block BLOCK_DEF
Priority 2 (prioil)  Not instantiated
Priority 1 (priohi)  Not instantiated
Priority 0 (priolo)  RdIdx: 0 WrIdx: 1    Pending: 1
#0 pending Valid Time: 0x1523c7a1dd6e1200    CmdType: flow
   Permanent: NO     Qty: 1    BLOCK_DEF --> MSG_A0
#1 empty   -
#2 empty   -
#3 empty   -
```

Listing 2.5: Output of dm-cmd queue inspection

carpeDM will list the content of all queues for the given block name, the result will look similar to listing 2.5. Because the DM was not told to run the schedule yet, we can see the flow command as still pending. We can also see that the change is temporary (not permanent), and the last column tells us that the flow goes from the default pattern exit (BLOCK_DEF) to the entry of pattern 'A' (MSG_A0).

This leaves the 'valid time' and 'Qty' properties. The valid time of a command specifies the WR time in ns *after* which a command is valid for evaluation, meaning the DM will not process it before that time. If the element at the front of a queue is not valid yet, no other queued elements will be evaluated neither. This also hold down the priorities: if the high priority queue is not empty but not yet valid, the low priority queue will also not be serviced. The repetition quantity (qty) specifies the number of times this element will yield the command it carries before it is exhausted and popped. In our example, the quantity is 1: the command will be executed once, then the containing element will be popped from the queue.

2.5 Static Commands

2.5.1 Concept

In the previous section, schedule behaviour was influenced solely from the outside. It is also possible to integrate commands into the schedule itself, allowing for a large number of new possibilities. This can be used as loops with initialisers (for), executing the following sequence n times. Another use is synchronisation, where one schedule is in a wait loop it will exit on the command from another schedule reaching the sync point. This approach can of course also be mixed with external commands, allowing for example for wait loops with a timeout.

2.5.2 Access Management

Static commands introduce a possible race condition within the DM, because the 1:1 relationship between command producers and consumers is no longer valid. There could be as as many producers per Queue as there are DM CPUs plus the host. This means that simultaneous access to a queue will create a conflict which must be handled. To prevent the race condition, a locking mechanism had to be introduced.

- Host: manages, sets and removes locks
  - always has priority when writing
  - must lock queue before write access
  - must verify DM CPUs obey set lock
  - does *NOT* manage producer–consumer constellations!
- DM CPUs: obeys locks
  - Locks are non-blocking
  - treat static commands to a write-locked block as Noop
  - skip queues of read-locked blocks, use default successor

Mechanism The command queue lock is a spin lock variant, using a hitherto reserved word within the block's data structure for lock flags and the command queue's read/write indices as indicators of activity. Locking of individual queues of a block is not possible, because all read or respectively all write indices are located in a single data word. Updating

the indices of different priorities from the host side would therefore still access the same word and cause a race condition. Lock flags are read-/write to the host and read only to the DM. Not all modules can be combined as producers and consumers of commands when sharing a block-/queue. There are several valid combinations which will produce orderly behaviour. As mentioned in the above list, it is not the responsibility of the host (ie. Generator FESA class with carpeDM library) to assign or validate the constellation of command producers and consumers per block. This task lies solely with schedule (LSA) and command generation (Director). The following constellations are valid:

Producer	**Consumer**	**Lock required**
Host	DM Cpu	RD*
DM Cpu	DM Cpu	–
Host & DM Cpu	DM Cpu	RD* & WR
EB Slave (UNI-GW)	DM Cpu	–
EB Slave (B2B)	DM Cpu	–

Sequence The host sets a lock, and checks the queue indices in regular intervals until no more changes are observed between checks. It is then certain that all ongoing DM actions (which might have been begun before the lock flags were visible) are concluded. The duration of host actions ms is three to four orders of magnitude longer than DM actions (µs), so a wait time in the low millisecond range between checks is sufficient. Once the lock flags are certain to be visible, the DM firmware will ensure that the locked block's queues are not modified. After the host has written to the queue, it clears the block's lock flags, allowing the DM to modify queues again.

2.5.3 Counter Loop Example

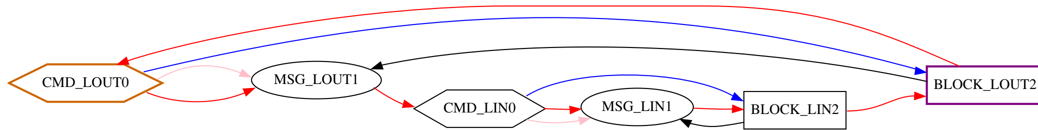


Figure 2.11: Counter Loop

```
1  digraph g {
2  name="CounterLoopExample";
3  graph []
4  edge  [type="defdst"]
5  node  [cpu="0"];
6    CMD_LOUT0    [type="flow",  pattern="OUTER", patentry="true", toffs  = 0, tvalid
       =0, qty=3, prio="0"];
7    MSG_LOUT1    [type="tmsg",  pattern="OUTER", toffs  = 0, fid=1, gid=4048, evtno
       =1, par="0"];
8    BLOCK_LOUT2  [type="block", pattern="OUTER", patexit="true",  tperiod
       =100000000, qlo="1"];
9    CMD_LIN0     [type="flow",  pattern="INNER", toffs  = 0, tvalid=0, qty=2, prio=
       "0"];
10   MSG_LIN1     [type="tmsg",  pattern="INNER", toffs  = 0, fid=1, gid=4048, evtno
       =2, par="0"];
11   BLOCK_LIN2   [type="block", pattern="INNER", tperiod=100000000, qlo="1"];
12
13   CMD_LOUT0 -> MSG_LOUT1 -> CMD_LIN0 -> MSG_LIN1 -> BLOCK_LIN2 -> BLOCK_LOUT2 ->
       CMD_LOUT0; //
14   BLOCK_LOUT2 -> MSG_LOUT1 [type="altdst"];
15   CMD_LOUT0 -> BLOCK_LOUT2 [type="target"];
16   CMD_LOUT0 -> MSG_LOUT1  [type="flowdst"];
17   BLOCK_LIN2 -> MSG_LIN1 [type="altdst"];
18   CMD_LIN0 -> BLOCK_LIN2 [type="target"];
19   CMD_LIN0 -> MSG_LIN1  [type="flowdst"];
20 }
```

Listing 2.6: Counter Loop

As described in 2.4.4 on page 29, commands come with a repetition quantity, specifiying how often they can be executed before they are popped from the queue. When the command is integrated into the schedule, this can used as a loop initialiser, similar to the head of a for-loop. Since each block has its own counter, there is no need for a stack to keep track of the variables. This allows nesting of several loops. The example in figure 2.11 sets up a nested loop, where the whole pattern runs infinitely, the outer loop executes 3 times and the inner loop executes 2 times per iteration. Line 13 nicely shows that the whole schedule is actually very simple, stringed together by the red default destination arrows like beads on a chain. Only once the commands get executed, there are several loops to go through.

It is obvious that the initialiser must only be called when there are not more repetitions of its command left, as it would otherwise flood the queue. This also means that you must not jump into or out of loops without flushing the corresponding queues.

2.5.4 Timeout Loop Example

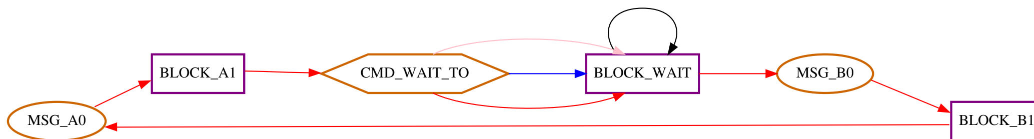


Figure 2.12: Timeout Loop

```
1  digraph g {
2  name="TimeoutLoopExample";
3  edge  [type="defdst"]
4  node  [cpu="0"];
5    MSG_A0      [type="tmsg",  pattern="A", patentry="true", toffs  = 0, fid=1, gid
       =4048, evtno=1, par="0"];
6    BLOCK_A1    [type="block", pattern="A", patexit="true",  tperiod=100000000];
7    MSG_B0      [type="tmsg",  pattern="B", patentry="true", toffs  = 0, fid=1, gid
       =4048, evtno=2, par="0"];
8    BLOCK_B1    [type="block", pattern="B", patexit="true",  tperiod=100000000];
9    CMD_WAIT_TO    [type="flow",  pattern="WAIT", patentry="true", toffs  = 0,
       tvalid=0, qty=50000, prio="0"]; //50000*20us = 10s
10   BLOCK_WAIT [type="block", pattern="WAIT", patexit="true", tperiod= 20000000,
       qlo="1", qhi="1"];
11
12   MSG_A0 -> BLOCK_A1 -> CMD_WAIT_TO -> BLOCK_WAIT -> MSG_B0 -> BLOCK_B1 -> MSG_A0
       ;
13   BLOCK_WAIT  -> BLOCK_WAIT [type="altdst"];
14   CMD_WAIT_TO -> BLOCK_WAIT [type="target"];
15   CMD_WAIT_TO -> BLOCK_WAIT [type="flowdst"];
16 }
```

Listing 2.7: Timeout Loop

A timeout loop is similar to the wait loop from 2.4.3 on page 26, it will exit on command, but it will also terminate after a given number of iterations (timeout). This can be achieved by using an initialiser to set up the time out, and then invert the exit logic: leaving the loop is now the default behaviour. Similarily, the command to exit is different: instead of issueing a flow command to leave the loop, we now issue a command to make the schedule stop staying inside the loop. Therefore, sending a flush command to the medium priority clearing the low priority (where the static flow went) will leave the loop before the timeout. The actual magic then happens in line 9, setting the length of the timeout to $qty \cdot period$.

2.5.5 Alternation with Default Pattern Example

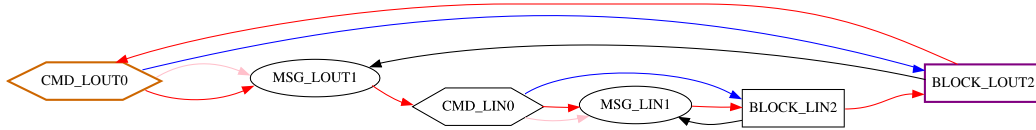


Figure 2.13: Alternating Counter Loop

```
digraph g {
name="CounterLoopExample";
graph []
edge  [type="defdst"]
node  [cpu="0"];
  CMD_LOUT0    [type="flow",   pattern="OUTER", patentry="true", toffs  = 0, tvalid
    =0, qty=3, prio="0"];
  MSG_LOUT1     [type="tmsg",   pattern="OUTER", toffs  = 0, fid=1, gid=4048, evtno
    =1, par="0"];
  BLOCK_LOUT2  [type="block", pattern="OUTER", patexit="true",  tperiod
    =100000000, qlo="1"];
  CMD_LIN0     [type="flow",   pattern="INNER", toffs  = 0, tvalid=0, qty=2, prio=
    "0"];
  MSG_LIN1     [type="tmsg",   pattern="INNER", toffs  = 0, fid=1, gid=4048, evtno
    =2, par="0"];
  BLOCK_LIN2   [type="block", pattern="INNER", tperiod=100000000, qlo="1"];

  CMD_LOUT0 -> MSG_LOUT1 -> CMD_LIN0 -> MSG_LIN1 -> BLOCK_LIN2 -> BLOCK_LOUT2 ->
    CMD_LOUT0; //
  BLOCK_LOUT2 -> MSG_LOUT1 [type="altdst"];
  CMD_LOUT0 -> BLOCK_LOUT2 [type="target"];
  CMD_LOUT0 -> MSG_LOUT1  [type="flowdst"];
  BLOCK_LIN2 -> MSG_LIN1 [type="altdst"];
  CMD_LIN0 -> BLOCK_LIN2 [type="target"];
  CMD_LIN0 -> MSG_LIN1  [type="flowdst"];
}
```

Listing 2.8: Alternating Counter Loop

While making two alternating patterns is a trival matter of setting their default destinations to each others entry points, alternating sequences mixed with the default pattern (d-A-d-B-d-A-d-B-. . .) are a more interesting case. Figure 2.13 shows how to achieve this with static flow commands inside the alternative patterns: The default pattern will run in a loop. If made to flow to pattern A, pattern A will then send a command to the default pattern to go to pattern B next. After one execution of the default pattern, B is executed, sending a command to the default pattern with pattern A as the successor, and so on. To leave this sequence, one would send a flush command to the default pattern at medium priority, after which the schedule would loop the default pattern.

Chapter 3

Offline Schedule Validation

3.1 Schedule Structure Validation

Schedules in carpeDM are built according to a given set of rules. Compliance is checked whenever a schedule graph in dot format is handed to carpeDM as a schedule action. Checks are made on different levels, such as node (unique name, consistent and complete set of properties, etc...), neighbourhood (types and numbers of neighbours a node can have), structure (sequences must have a terminating block, etc...)

The action a schedule comes with (add, keep, remove) is also important as context to determine if the given schedule is valid. For example, removing pattern X from the global graphs existing pattern set of X,Y is fine, but adding a redundant X is not. Likewise, adding the pattern Z to the set of X,Y would be okay, removing the (not yet existing) Z from X,Y is not.

carpeDM therefore validates all given dot graphs for their purpose and aims to give an elaborate reason in its feedback when a schedule is rejected. This chapter presents the details of the schedule validation scheme and lists part of the implementation.

3.1.1 Validation on creation

Following is the list of rules carpeDM applies when validating schedules. The ruleset is split in a part applying to "real" nodes and a part only applying to meta data nodes, which are not included in the standard schedule output. Those nodes contain data for internal use by carpeDM.

Rules for real nodes

- Sequences
  - Real nodes are timing messages, commands and blocks
  - A sequence is a set of real nodes connected by default destination edges
  - Sequences can be connected to other sequences by default or alternative destination edges
  - The maximum number of alternative destinations is 9 (subject to change in future)
  - All sequences must be terminated by a block
  - All real nodes except blocks must have a default successor
  - Only blocks are allowed to have themselves or none (idle) as default successor
  - The shortest possible sequence is a lone block
  - Sequences can form infinite loops
  - Time offsets within a sequence must be in ascending order
  - The max. time offset in a sequence must be less than its block's period
  - Sequences connected by default or alternative destination edges must reside on the same CPU
- Patterns
  - Patterns must have exactly one entry and one exit point (might be subject to change in future)
  - Pattern entry points can be timing messages, commands or blocks
  - Pattern exit points must be blocks
  - All of a patterns nodes must reside on the same CPU (might be subject to change in future)
- Branching
  - Branching requires a block with at least one queue
  - Stopping (unlinke aborting) is branching to idle
- Commands

- Commands always target blocks, but the target can be empty
- All commands can target blocks on own or other CPUs
- Flow command destinations must either be real nodes or empty
- Flow command destinations must be on the same CPU as the target block
- Flow commands cannot initialise a loop they are a part of

Facts about meta nodes

- Only blocks can have meta nodes, allowed are 0-3 queue buffer lists and 0-1 destination list (subject to change in future)
- Only queue buffer lists can have queue buffers, 2 are currently mandatory
- Management nodes contain compressed node names, group memberships and/or covenant data. Cannot be created manually

Guidelines

- To add queues, just list the priorities you want. carpeDM will handle the overhead of meta node creation for you
- If a block has exactly one successor, don't add queues, this saves space
- carpeDM will automatically add a destination list to a block if any alternative destinations are present
- When using commands in schedules, 99.9% of the time you will need them ASAP (vabs=true, tvalid=0)
- *Only define meta nodes manually if you know exactly what you are doing!*

3.1.2 Validation on change

Rules for *add*

- An *add* is a list of nodes and edges to be added and is a dotfile by itself
- You cannot overwrite existing nodes, edges or their attributes using *add*. Remove them first, then add new versions
- If the addition is connected to existing nodes, only specify the edges to those nodes, not the nodes itself
- You cannot add outgoing edges to active schedules except alternative destinations. See chapter 4 for details on online verification

Rules for *remove*

- A *remove* is a list of nodes and edges to be removed and is a dotfile by itself
- All nodes listed for *remove* must exist in the DM graph
- All edges leading in or out of removed nodes will also be removed
- You cannot remove nodes from an active schedule. See chapter 4 for details on online verification

Rules for *keep*

- A *keep* is a list of nodes and edges to be kept and is a dotfile by itself
- A *keep* is a *remove* of the difference set of the *keep* set and the DM graph
- All nodes listed for *keep* must exist in the DM graph
- You cannot keep edges without keeping their nodes
- All edges leading in or out of not-kept nodes will also not be kept
- You cannot not-keep nodes from an active schedule. See chapter 4 for details on online verification

3.1.3 Intentional late message generation

It is possible to create late timing messages on purpose for debugging. This can be achieved by specifying negative time offsets for individiual timing messages. The negative offset must have an absolute value greater or equal the normal (as in "fitting into the ascending sequence") time offset. Such a debugging schedule is in violation of the rule set and will be rejected by carpeDM. To force the acception, you must set the force flag in the Generator FESA class (on CLI, run *dm-sched* with "-f" option).

```
$ dm-sched <eb-device> add -f <late-message-dot.dot>
```

3.1.4 Summary

The offline verification rule tables and algorithms make sure only valid schedule data will be accepted for upload to the DM. All of the rules listed above except the ones about active schedules in subsection 3.1.2 are independent of current DM activity and therefore evaluated "at compile time" of the schedule. The only exception is the use of absolute time values within schedules, which could become obsolete before upload is achieved. There is currently ($\leq$ v0.27.1) no safeguard against the use of stale absolute times. Almost all of the rules are not solely good practice, but absolutely necessary to achieve expected behaviour of the DM firmware. However, for certain debug cases, it is possible to bend the rules somewhat without causing havoc in the DM hardware.

Chapter 4

Online Schedule Modification and Safeguards

4.1 Overview

4.1.1 Problem Definition

During runtime, schedules often need to be modified. A trim is a perfect example of a measuring loop, which will iteratively change schedule data. There two systems which simultaneously access the DM's memory – the high level host side and the DM realtime system. Any schedule data which is currently in use by the realtime system cannot be modified by the host without causing undefined behaviour. It is therefore important to determine wether and when modifying a schedule is safe.

Data basis Knowing which schedule data is actively used in the DM is therefore of paramount importance in the decision wether a schedule can safely be modified. Since the host system can only ever be broadly aware what the DM RT system is executing at any given time, discerning between active and inactive schedule data is not as trivial as it might sound. We will refer to all objects (schedules, paths, edges, nodes) that must not be used during safe manipulation as *critical*.

The available data consists of the complete schedule graph, the content of all command queues and the cursor positions, which mark which nodes of which the schedules the DM was executing. Because a lot of this data in the DM RT system can change during processing time and even during the data acquisiton itself, the memory image obtained by the host is suffers from a sort of "motion blur", which must be considered. A valid

approach must divide the data into conditions proven to be present, conditions proven not to be present and use the worst possibly outcome for any ambiguous cases.

Testing for Safety Safety means guaranteed inactivity. In order to give a guarantee on the basis of an inconsistent data set, all time factors (execution times, race conditions, atomicity...) must be eliminated from the verification process. It is easy to see that a schedule is active if a cursor is currently pointing to one of its member nodes. Considering the "motion blur" and our own processing time, the cursors might also already have left the schedule in question - the case it ambiguous and the worst case to be used is the cursor still being inside. Likewise, seeing the cursor outside a schedule is no guarantee for its inactivity. A cursor might well have entered it again just after we had a look. . .

4.1.2 Possible Approaches

Several approaches to modifying a schedule safely were investigated, all of which have pros and cons in terms of the dimensions Safety – Speed – Low Memory Req..

- The first is the first write a new version of the schedule in question, command the DM tw switch over to it. After assertaining the DM has left the obsolete version, it can be removed. From a runtime perspective, this is the safest and fastest method. However, there are drawbacks: Because essentially a copy is created (albeit slightly modified), twice the space of the original is required. And because it is uncertain when the DM will have left the obsolete version (possibly hours), this requires an asynchronous garbage collector on the host side.

- The second method is as safe as the first, more memory efficient, but also often much slower. The DM is redirected to another (safe) schedule, and once it is certain the DM has left the obsolete schedule and has no possibility of re-entry, it is removed. The new version is then written and the DM is commanded to use the new version. This is very space efficient, but verifiying that the schedule to be removed cannot be entered again is very challenging. Waiting for the DM to leave areas that could reconnect to the obsolete schedule can take up a lot of time (in the case of ESR, this could take hours). The details are described in subsection 4.2.

- The third method is a hybrid approach. Sacrificing some safety margins allows combining the speed of the first approach with the low memory requirement and simple management of the second. By extrapolating future DM behaviour from command queue content, it is possible to eliminate certain static paths from considerations. This often allows safe removal of the schedule almost immedately, but comes at a price. The prediction will only hold true if the content of the involved command queues is not changed. The requesting party therefore enters a covenant with carpeDM once it removes the schedule in question. The covenant contains a list of command queues and their priorities which must not be modified or preempted, otherwise there will be undefined behaviour. Once all critical queue content is processed, the covenant is fulfilled. Details are described in subsection 4.3.

- The fourth method is radically different to the other three, as it modifies queues within an active pattern. To do this, lock bits are set for the block, ordering the DM to refrain from reading or writing queues associated with this block. This by far the the fastest method to communicate changes to the DM in a safe manner, but also the most invasive. The problem can be seen in the definition of "safety". The approach will not cause undefined behaviour, but can suppress commands originating within the DM. 4.3.

4.2 Equivalent Static Model

The possibility of future cursors positions being inside the critical schedule makes it necessary to inspect all possible paths leading into the schedule (that it, to its entry point). A time invariant representation of the schedule with all of its static and dynamic links must be created and checked against the cursor positions. If a cursor is within the schedule or if a path from a cursor to the entry point exists, the schedule must be seen as active which makes removal unsafe - it is not to be touched. Likewise, if there is no cursor inside and no path to the entry point can be found, the schedule is inactive and can thus be safely removed. To draw any kind of usable conclusions, the director must remain silent once the verfification is in progress, so no more commands are entering the system. Any asynchronous external devices issueing commands such as the UNIPZ gateway must only write to their own uncritical schedules or remain silent.

Handling inconsistency in memory snapshots Reading out the data from several processors will lead to an inconsistent image of the current DM state. Depending on the type of objects, there are different appropriate methods to deduce facts about the state from of the available data.

- *Default Successor Edges* are definite if the edge's parent is not a block with commmand queues. However, if that is the case, then the default successor is ambiguous and queue content must be considered. If the default successor of the block can be changed temporarily or permantently by its queue content, both the old and the new edge must be used.

- *Qeue Content* is handled by both the DM and the host. They use the read and write indices of the queues' ring buffers to synchronise their access. Only the host can write new elements and modifiying the indices is always the last action in any queue access for both sides. When reading the verification data, the first action of the host is always to get the current WR system time. All commands written by the host must bear a current valid time (the moment in time after which the command is valid). This means that we can tell by the indices which commands are definitely consumed already and by the valid times which commands are definitely not consumed yet. All others must be seen as "possibly consumed", which means using their worst possibly impact.

- *Cursors* are the most "blurry" data objects which are read during verification. A cursor therefore can be seen not as a single node, but as a subtree of nodes originating at the observed cursor location and spreading to all reachable nodes. If the entry point would be found within the set of nodes formed by the cursor subtrees, it follows that the schedule is active. This makes the approach independent of the progress of the cursors since observation.

  The used implementation is in fact the inversion of the cursor subtree approach just described: a single subtree is constructed from the entry point of the schedule backwards, intersection with any cursor node shows the schedule is active.

4.2.1 Path Analyses

Static The most basic form of analysis considers only static (default successor) paths, all other forms of connections are removed to simplify the graph. The reverse tree originating at the entry of the critical schedule is itself critical. If a cursor is inside the tree, it can reach reach the entry, making the schedule active and thus unsafe to modify.

Dynamic Static path analysis would not yield correct results in the presence of commands changing the flow through the graph at runtime. To cover this, the queue content must be analysed for flow commands. This not only covers the dynamic commands actually present in queues at readout, but also those which can be generated by resident flow commands. Those are flow commands which are part of the schedule.

Virtual paths To still allow the use of graph algorithms, dynamic changes must be modeled in a way that can be handled by such methods, which means a static equivalent graph using virtual edges. These show all possible paths resulting from commands and can be analysed for intersection just as the static graph. However, there is a corner cases to consider, which is resident flow commands. They also must be represented by a virtual edge, but only if the block the virtual edge would originate from is within the limits of nodes which can be reached by a cursor.

Iterative construction of virtual paths Dynamic links only become real if the block is actually ever visited by a cursor. Therefore, not all queue

content is worthy of consideration, all dynamic commands and static command generation which cannot be reached by any cursor are of no consequence. To be considered as valid virtual path(s), the reverse tree originating at any block with generated commands must have a connection to at least one cursor.

Reverse Tree and Intersection After the removal of all non-path edges and the addition of all virtual paths, the result is a equivalent static graph model (ESM). Using the entry point of the schedule to be removed as the start, the reverse tree is then constructed, going against edge direction and mapping all nodes connected by default or virtual paths. Loops in the tree are detected. Furthermore all member nodes of the schedule to be removed are added. This forms the complete set of nodes which have a connection to the entry point. If there is an intersection of this set with the set of cursors, the schedule is not safe to remove.

4.2.2 Orphaned command handling

When removing a schedule, a severe side effect can occur in inactive schedules. The queued commands of inactive schedules are ignored in the safety assessment, as they do not have an impact on the outcome. It follows that all flow commands pointing from inactive schedules to the critical schedule would become orphans when the critical schedule is removed, as their destination suddenly ceases to exist. If the inactive schedule is ever reactivated and its queue processed, this will cause undefined behaviour. Simply removing orphaned commands from queue buffers is not an option though, because the ring buffers cannot contain gaps. While this problem could be overcome by moving all other elements, there is a more elegant solution available. By setting the repetition counter of orphaned commands to 0, the command is no longer invalid but instead acts as a Noop instruction. It will then be normally processed and popped from the queue without further consequences.

4.2.3 Visual Reports

The implementation is capable of creating visual reports for the reasoning behind a found decision. This is achieved by colorisation of all traversible paths to entry points of the schedule to be removed in red, as unsafe. This is handled inside the static equivalent model, the worst case scenario where all possible future connections are treated as existent. The execution

cursors, or rather, the last known node positions of them, are coloured in as well. If any happens to be inside an unsafe area, the conclusion is that a cursor could currently be inside the schedule to be removed or enter it in the near future. This allows humans to easily follow the connections between nodes with their eyes, even in large patterns. The colorisation gives a simple yet effective distinction of territory.

4.3 Enhanced Equivalent Static Model (aka "crystal ball")

4.3.1 Problem Definition

The observed trouble with the approach described in section 4.2 is the blocking nature of the process. The resulting wait times can encompass all schedule duration encountered before. In the case of long schedules ($T \gg 100\,\mathrm{ms}$), this becomes a severe hindrance for trimming, not to mention the de facto freeze the ESR's schedules with a duration of hours or days would cause. This effect will always come into play if there are several long schedules preceding one that is used to redirect the DM away from the schedule to be removed. Before the command for redirection is not consumed, there is no possibility of guaranteeing safety. An approach to make this verification non-blocking had to be found.

4.3.2 Contextual inconsequence of default successors

If a default successor edge is a connection between a cursor and the entry point, it is tipping the scales toward the verdict "unsafe". The main issue hinges on the fact that dynamic changes to default successors are executed at the time their block is visited by a cursor. This means the change can come into effect a long time after the corresponding command was issued, causing the aforesaid wait times. The crucial question is therefore if the change to the default successor edge can already be considered a fact at the time the command is present, thus skipping the wait time to command processing. Found optimisations are expressed in the resulting ESM by removing the corresponding default edges and replacing them with dashed auxiliary edges.

Continuous successor override A default edge can be optimised away if the DM can never take that path. This is the case if the default successor

is either permanently changed to a safe schedule or changed to idle. In the latter case, permanent or temporary change type is irrelevant, as a cursor cannot return from idle without outside intervention. We will thus also consider idle as a permanent change.

However, there may be more than one command present in the queue(s), and each visit by a cursor will only consume one charge of the top element. If the default destination is never to be used, it means that it must be constantly overridden by queued commands until the permanent change is reached. Since only flow commands can override the default, it follows that the presence of any other command type before the permanent change forbids optimisation.

Command order of precedence The order in which commands are executed depends on the priorities of the queues they have been written to. Highest priority present is always emptied first, the maxmimum queue length is 12 elements (4 x High, 4 x Mid, 4 x Low). It follows that once an optimisation is found and deemed a contribution to a "safe" verdict, any preemption of the containing queue is forbidden. This includes filling higher priorities with non-flow commands and the static flush option, which allows the host to asynchronously clear a queue.

4.3.3 Finding contributing optimisations

To find the queues to be protected for a safe schedule removal, it is necessary find which of the optimisations contributed to the positive outcome. The basic assumtpion is that any path leading to the entry point containing optimised edges can be optimised, it depends on all commands causing the traversed optimised edges the corresponding queues must therefore be protected. However, if such a path can be circumvented by another, the corresponding queues is irrelevant and should not be protected. A scheme had to be found which not only maps the optimisation depencies, but also preferably works with the intersection set test introduced in 4.2.1.

Dependency Mapping The dependencies are the traversed optimised edges in critical paths (as in leading to the entry point). This is ultimately equivalent to the queues to be protected, which means the identifier of the block at the origin of the optimised edge. Because critical paths can depend on one another or be mutually exclusive, it is necessary to analyse all paths in parallel and accumulate their dependencies. To achieve this, the crawler once again creates the reverse tree originating from the entry point

and propagates an individual set of dependencies along each branch, the content depending on the encountered edges. This means each node is assigned a set of all depencies encountered on route from the entry point to itself. As long as a default or virtual edge is traversed, a null identifier is added and propagated. When an optimised edge is traversed, the identifier of its source block is added to the dependency set and the propagated value is changed to the block identifier. This means only paths which depend on at least one optimisation will not carry null identifiers. If a node is reachable over multiple paths, all propagated identifiers will accumulate there.

Enhanced Intersection Test After the dependencies are mapped, we can once again check for the intersection between the reverse tree and the cursors. This time, however, there is an added twist to the process: Only the tree nodes whose dependency set contains a null identifier are considered when testing for safety, as they represent the critical paths. For all other members of the intersection set, their individual dependency set will show which queues must be protected in order to exploit the optimisation.

4.3.4 Covenants

The union of all dependency sets of non-critical intersections form a set of important commands. It allows the user to immedately remove the schedule in question in exchange, provided he does not preempt or delete any of the listed commands. The list is employed as covenants with the user (LSA / Director). Removing (by *remove* or *keep* command) the schedule will call the safe removal verfication and seal all associated covenants on success. Requesting verfication on its own has no lasting consequences.

Covenants contain the block identifier, the queue priority and a checksum of their command. They are written to the DM memory as part of the management data and are automatically checked with each download operation and updated with each upload. If the contained command has been consumed, a covenant has been fulfilled and is removed. carpeDM will reject requested operations which would break a covenant in order to prevent undefined behaviour of the DM. Override can be forced if deemed necessary at the user's own peril. Like with all forced operations, the circumstances of the incident will be reported for later review.

4.3.5 Handling cursor to queue race conditions

Validity of Commands When interpreting the DM's memory image, we have talked in depth about the blurriness of observed cursors. Yet another side effect of this comes into play when using the enhanced safe2remove alogrithm, because before now, all that was considered were worst cases. When interpreting queue content, this meant every element could be valid and was therefore added as an edges. This was the worst case, as the possible maximum of edges which could lead into critical areas were considered, no matter the order of execution. Since incrementing a queues read index is atomic and always the last action executed by the DM firmware before proceeding, it is known for certain that all elements seen as popped are indeed consumed. There might be less unconsumed elements than obeserved, but certainly not more, thus erring on the safe side.

Now, however, the enhanced approach goes in the other direction, trying to shave something off the conservative first assessment. The question is therefore: Under which circumstances can queue elements be ignored?

When to ignore queue elements There are some possible answers to this, one might be that everything on lower priority than a flush command can be treated as void, for example. This is something not currently used, but should be investigated in the scope of future work (as of 22.10.2018).

Yet another (implemented) approach is comparing the valid-timestamp of an element to the time at which the cursors were read. If read time is lower than an encountered valid time, the corresponding element is not available now and might still not be when the cursor arrives at this block. A valid-timestamp $\leq$ read-time is thus a necessary condition to readjust the conservative assessment by overriding a critical default destination.

As an example, let us assume the element at the front of the queue is a flow command, changing the block's critical default destination to a safe alternative, so it would make removal of the schedule safe. But because it's valid time is not yet reached, the outcome is uncertain - the command might or might not override the default destination. Because the analysis is not to rely on execution times, this flow command cannot be used as justification for an optimisation at this point.

4.4 Summary

A reliable means of telling if a schedule branch is inactive is a core requirement for safe changes to schedules during runtime. Doing such changes

blindly on an active schedule would run into several race conditions with severe consequences in terms of pointer errors and following memory corruption. A verification algorithm able to tell if a schedule is inactive and therefore safe to remove was created. The second and third approach listed in **??** were implemented.

Equivalent static model This approach creates a time-invariant equivalent model, the ESM, from the original graph. In combination with the cursor positions in the DM, a safety assessment for the schedule removal is possible. The ESM is obtained by stripping the original graph of all but the default successor edges and adding virtual paths representing all flow commands in queues. Further virtual paths are iteratively added for all resident flow commands reachable by the current iteration of the model. The virtual path scheme provides a representation of flow commands which is independent of the time the command queues were observed. In the resulting ESM, the reverse tree originating from critical schedule's entry point can be mapped. If an intersection with the set of cursors is found, the schedule is active and therefore unsafe to remove. This result is independent of the cursor's progress since their observation time.

The approach was proven to be valid by simulation for 3 fully connected patterns (full coverage). The drawback was, as estimated, the blocking behaviour of the algorithm, producing possible wait times for a positive result in the dimensions of the longest pattern involved.

Enhanced Equivalent static model The enhanced verification algorithm was based on the ESM, augmenting it by replacing critical static links in the ESM with safe dynamic overrides. This requires a dynamic, permanent override (flow command) to a safe alternative destination to be present at the front of the queue or preceded by an unbroken chain of safe temporary overrides. Either allows immediate removal of the schedule in question; the involded queues must not be preempted until the crucial override to safety is executed. These promises have been named *covenants* and are automatically managed and enforced by carpeDM.

The algorithm was also proven to be valid by simulation for 3 fully connected patterns (full coverage). Wait times from the first approach were eliminated in exchange for compliance of the user with all active covenants. This is the standard for carpeDM $\geq$ v0.18.0.

Chapter 5

Offline Resource Analysis

5.1 Memory Load

5.1.1 Problem Definition

Memory is a sparse resource in the DM, as it completely resides within the FPGA. Each CPU is assigned its own dual port memory containing all schedules this CPU executes. CPUs can technically also execute schedules residing outside their own memory. However, shared bus access is severe a bottleneck and source of non determinism, so this approach should be avoided at all costs. The consequence is that the assignment of schedules to CPUs has to be carefully planned to maximise resource utilisation.

5.1.2 Graph Data

5.1.3 Meta Data

The overhead data present can be divided into two categories.

The first is directly linked to the schedule on a local basis and will be called schedule meta data. Their meta data forms distinct entities per node and can be read just like any other schedule node. These come in the following varieties:

- Alternative destination list
- Queue buffer list
- Queue buffer

The second type contains meta data important to all schedules as a whole and is referred to as management data. Contrary to schedule data,

management data is compressed and the archive is spread across the linked list of all management nodes. It can therefore not be interpreted on a per node level. Management data follows the minimal structure of nodes(type field, next ptr), but is only ever read by the host, never by the DM embedded system. It contains the following overhead information:

- Node relationship table
  - Node name
  - Pattern name
  - Node is entry to pattern
  - Node is exit of pattern
  - Beamprocess name
  - Node is entry to beam process
  - Node is exit of beam process
- Covenant table

5.1.4 Load Balancing

carpeDM $\geq$ v0.16.0 does auto-balance management data, but not schedule meta data, over all CPU RAMs. The currently is no memory load balancing for schedule data, all nodes are directly assigned to a CPU/RAM via a tag in their definition. This will be subject to change, but requires an a priori analysis of schedules, guaranteeing processor load to stay $\leq 100\%$. Since this is not implemented yet, an automatic assignment to CPUs is not sensible at the moment. See section 5.2 for details on network calculus based load analysis.

5.1.5 Summary

5.2 Network Traffic

5.2.1 Problem Definition

CPU Performance

Network Performance

5.2.2 Introduction to NC

5.2.3 Introduction to DISCO DNC

5.2.4 DM to Endpoint Model

5.2.5 Arrival Curves from Schedules

5.2.6 Verification Process

5.2.7 Load Balancing

5.2.8 Summary

Chapter 6

Theoretical Model

6.1 Overview

6.1.1 Motivation

This chapter investigates one of the most important aspects of this CS – the conditions under which it can be guaranteed to work deterministically. The DM is the head of an alarm based CS which provides a certain degree of freedom when choosing a "safe" lead for transmission. There are limiting factors though:

- Target control loop speed
- Available computing power
- Available network bandwidth

It is therefore necessary to create a model of the CS, which verifies if a chosen set of machine schedules can be scheduled within CPU utilisation $\leq 100\%$, does not generate traffic $> 1\,\mathrm{Gbit\,s^{-1}}$ (including overhead) and does not exceed the target delay. This goal is made more difficult by the possible need to change machine schedules during runtime. This will happen whenever interlocks and beam requests from experiments need to be serviced.

The purpose of the model is to provide a guarantee that a given set of messages can be delivered on time.

6.1.2 Choice of Implementation

After a period of research in classical queuing theory **yue_advances_2009**, **daigle_queueing_2005** the conclusion was reached that queuing theory

is not well suited for the task at hand. Queuing theory is very generic, requiring a lot of the mathematical elements to model this specific problem to be developed first. Queuing theory is also more focused on throughput and probabilities, rather than determinism and providing delay bounds. Attempts in modelling the DM system using queuing theory turned out to lead into a lot of dead ends, the model never quite matched the prototype. Without an expert level knowledge in the field, there seemed to be little chance of accurately model a system as complex as the DM in queuing theory.

The author's specialisation in electrical engineering is communications, which is probably the main reason why *Network Calculus (NC)* had to be encountered at some point in the search for a suitable tool to solve this particular problem. Parts of the problem in modelling the DM were exhibiting a striking resemblance to problems quite common in system theory. Modelling machine schedules, in a way that would make their superposition and shifting in time manageable, showed a lot of the hallmarks of signal processing. Expressing the burstiness of a schedule over time as a function of frequency in a spectrogram seemed natural, just as the idea to smooth bursty flows by a filter element did.

After deeper investigations of the work of **cruz_calculus_1991** (1991), and later **thiran_network_2001** (2001), NC presented itself as an ideal tool for the problem at hand. NC is a mathematical framework to model concepts from system theory in a networking context, with focus on deterministic behaviour and bounds.

6.2 Introduction to Network Calculus

6.2.1 Overview

System Theory for Signal Processing Signal processing is a very important field in electrical engineering and computer science. In most electronic devices signals must be generated, shaped, filtered or distorted. Sheath current filters suppress low frequency humming in a sound system, a blur effect removes the high frequencies from an image, to just name a few examples.

Complex behaviour in system theory is modelled by the concatenation of basic elements. These are small two-port networks, for which the signal transfer functions can be calculated. System theory provides the necessary mathematical tools to put these elements in series or parallel and so enables the calculation of complex signal transformations.

Computer Networks Another application of system theory would be modelling the traffic in computer networks. Similar to signals, network traffic can be shaped, filtered, split or joined, but the application of classical system theory is rather tedious. Specialisation within queuing theory has evolved to deal with the flow in computer networks, focusing on optimising throughput.

Network calculus NC is an approach that applies system theory to deterministic queuing systems found in communications, such as computer networks. Contrary to traditional system theory used for electronic circuits, NC employs a different set of algebra, the Min-Plus Dioid (addition becomes computation of the minimum, multiplication becomes addition). The approach is aimed at understanding and modelling fundamental properties of networks, such as delay or buffer requirements, scheduling or window flow control. The focus of NC lies on worst case analysis in order to provide guarantees for a communication system.

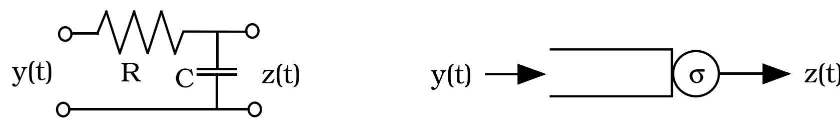


Figure 6.1: Equivalency: System Theory Low-Pass
and NC Shaper **thiran_network_2001**

As an example, the similarity between an Resistor-Capacitor (RC) low pass filter in system theory and a rate-limiting shaper node (server) in NC is shown in figure 6.1. The two-port filter network on the left transforms an incoming analogue signal by applying the convolution with the circuit's impulse response. In electrical engineering, a two-port network has a transfer function, which defines the output voltage in relation to the input voltage. In NC, input- and output-"signals" are cumulative flows. This means the cumulative sum of units of data (bits, words, packets, etc.) over time.

The similarity between a signal filter and a shaping node becomes more apparent when considering a constant rate of arriving packets is equivalent to a *frequency* of packet arrivals. It therefore follows, that a shaper imposing a maximum rate (frequency) is low pass filtering the packet flow (signal).

Example NC focuses on guarantees, it shows the bounds for maximum delay and backlog that a flow can experience. A convenient property of

NC is the minimal effort necessary to obtain backlog and delay values. As figure 6.2 shows, any input flow $y(t)$ passing a node produces an output flow $z(t)$ for which $z(t) \leq y(t)$ holds true. At any point in time, the current backlog can be determined by the vertical deviation of the flows. The current delay can be determined by calculating the horizontal deviation. Finding the respective maximum values by applying the supremum is then trivial.

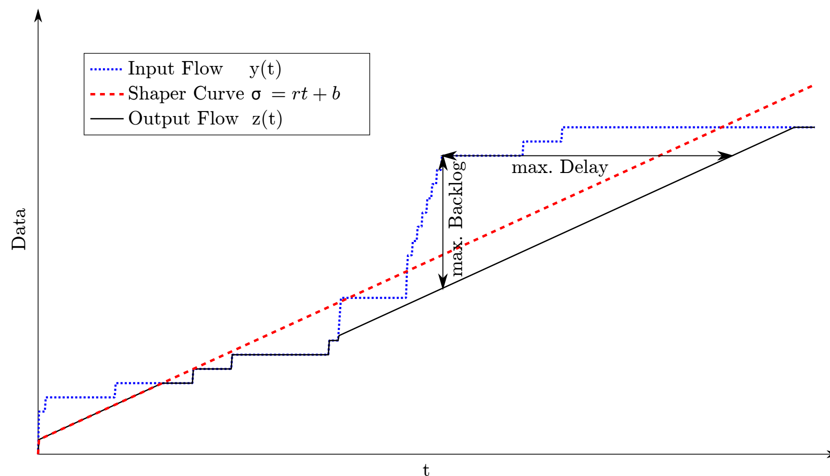


Figure 6.2: Flow passing through a Shaper

6.2.2 Network Calculus Core Concepts

While flows are defined as the cumulative sum of data over time, NC defines systems in terms of arrival curves and service curves. Figure 6.3 shows two related examples.

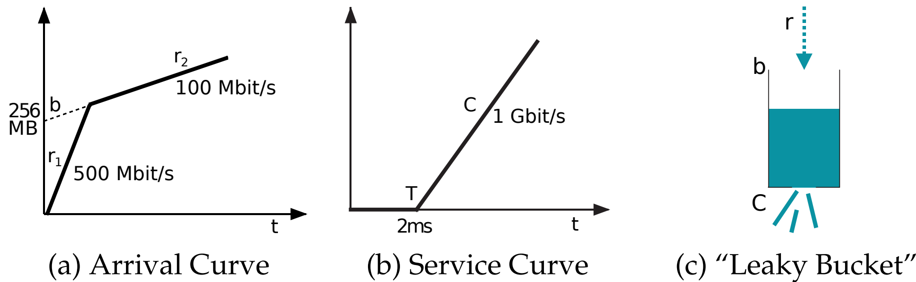
Arrival Curves Describe sets of constraints that govern the input flow's behaviour over time. These curves are usually piece-wise linear, concave functions. Their slope describes the maximum allowed rates, their vertical offset the burst tolerance.

An arrival curve could state that an incoming flow is allowed a peak rate of no more than $500\,\mathrm{Mbit\,s^{-1}}$ up to an input buffer size of $256\,\mathrm{MB}$, afterwards it must fall back to a sustainable rate of $100\,\mathrm{Mbit\,s^{-1}}$ until the buffer's fill level is lower (figure 6.3a).

Service Curves These are the counterpart to arrival curves, they describe the service a system offers to an input flow. Their horizontal offset is describing the amount of time lag packets experience, their slope describes

the minimum rates. To again provide an example, a server's minimum service curve could state that it will delay packets for at least $2\,\mathrm{ms}$ and can handle traffic up to a rate of $1\,\mathrm{Gbit\,s^{-1}}$. These curves are usually piece-wise linear, convex functions (figure 6.3b).

One of the core constructs of NC is a node behaviour called the "leaky bucket controller". The analogy is simple: Consider a water bucket, able to hold an amount of water b, leaking with a constant rate of C. It can handle one or more of gushes of water of arbitrary volume, up to the capacity of the bucket. Once the bucket is full, it is easy to see that there is a maximum rate at which one can add more water without overflowing, which is $r \leq C$ – a system with buffer size b, input data rate r and output data rate C. This is equivalent to featuring an arrival curve $\alpha = \upsilon_{b,r} = rt + b$ and a minimum/maximum service curve $\beta = \lambda_C = Ct$ (figure 6.3c).



(a) Arrival Curve (b) Service Curve (c) "Leaky Bucket"

Figure 6.3: NC Examples

Output Arrival Curves Arrival and service curves put constrains on input and system behaviour, but to give a guaranteed flow behaviour, the output of a system also needs to be constrained. The matching instrument is called an output arrival curve. It is similar to an arrival curve, but describes the flow *after* it has traversed a node, having experiencing its service. It is used to define the input constraints for the next downstream node. As an example, consider a periodic input flow, sending with a rate of $1\,\mathrm{Gbit\,s^{-1}}$ for 20% of the time and idle otherwise. If the node provides a service of $250\,\mathrm{Mbit\,s^{-1}}$, the output arrival curve would also be a periodic flow, sending at $250\,\mathrm{Mbit\,s^{-1}}$ for 80% of the time.

Shaping Curves are the application of service curves to form a flow. If a node forces a flow to conform to a specific target arrival curve, it is called a shaper. More details on shapers can be found in subsection 6.2.4.

Bounds The main goal of NC is to provide bounds on delay, backlog and output, in order to give guarantees. Bounding the end-to-end delay a flow experience is for example necessary when calculating if the lag for an individual VoIP connection will always stay low enough not to impair a conversation. Bounding backlog, for example, is important for calculating the size of memory a router will need. And lastly, the necessity for bounding an output flow, which is explained under "output arrival curves".

6.2.3 Mathematical Background

Most of the theory on Network Calculus in this chapter stems from **thiran_network_2001**'s text book on the subject **thiran_network_2001**. This introduction will focus on the application of the presented theorems and limit the formal proof and mathematical background to the necessary minimum. After the work of **thiran_network_2001**, NC has forked into the direction of stochastic applications **jiang_basic_2006**. This is of no further interest to this case study, as the CS must be deterministic. Some of the most interesting advance in the field of deterministic NC, as well intriguing problems and tricks of the trade, were gathered from the post-2006 publications of **schmitt_comprehensive_2007**, as well as Fidler and most recently, Bondorf **schmitt_comprehensive_2007**, **fidler_way_2006**, **schmitt_delay_2008**, **bondorf_discodnc_2014**, **bondorf_improving_2016**.

Min-Plus Algebra NC uses a different algebraic Dioid (similar to e.g. Boolean algebra, which replaces arithmetic operations by logic), which replaces addition with computation of the minimum and multiplication with addition. The two most important equations describe the convolution operations, similar to standard system theory. They are defined as

$$\begin{aligned} &\text{Min-Plus Convolution} \quad && f(t) \otimes g(t) = \inf_{0 \le s \le t} \{f(t-s) + g(s)\} \\ &\text{Min-Plus Deconvolution} \quad && f(t) \oslash g(t) = \sup_{s \ge 0} \{f(t+s) - g(s)\} \end{aligned} \tag{6.1}$$

Max-Plus Algebra The corresponding max-plus operations are also listed, not only for the sake of completeness, but because max-plus deconvolution will become useful when finding a curve providing a lower bound to a function. The concrete application will be shown in conjunction with

scaling operators in section 6.2.4.

$$\begin{aligned} &\text{Max-Plus Convolution} \quad f(t) \,\overline{\otimes}\, g(t) = \sup_{0 \le s \le t} \{f(t-s) + g(s)\} \\ &\text{Max-Plus Deconvolution} \quad f(t) \,\overline{\oslash}\, g(t) = \inf_{s \ge 0} \{f(t+s) - g(s)\} \end{aligned} \tag{6.2}$$

Curves We will follow the convention of marking output related curves and flows by appending an asterisk. Arrival curves (if not otherwise stated, an upper bound) are denoted by the letter α (and therefore, α^* denotes an output arrival curve). For service, there exists a maximum service denoted as γ, which is useful to calculate buffer sizes, and minimum service denoted β, which is used to calculate delay. Shapers are always denoted as σ. The definitions of all introduced curve types are as follows:

$$\begin{aligned} &\text{Input-Output Relation} \quad R(t) \ge R^*(t) \\ &\text{Max. Arrival Curve} \quad R(t) - R(s) \le \alpha(t-s) \\ &\text{Max. Service Curve} \quad R^* \le R \otimes \gamma \\ &\text{Min. Service Curve} \quad R^* \ge R \otimes \beta \\ &\text{Shaping Curve} \quad R^* = R \otimes \sigma \le (R \otimes \alpha) \otimes \sigma \end{aligned} \tag{6.3}$$

Bounds The following equations concern the "three bounds", as **thiran_network_2001** called them: backlog, delay and output flow. Backlog and delay can be directly calculated from the difference between input and output flows (see figure 6.2), as well as from arrival and service curves. It is easy to see here, that a node with an arrival curve, of a higher continuous rate than the service curve, cannot have bounds.

$$\begin{aligned} &\text{Flow Backlog} \quad b(t) = R(t) - R^*(t) \\ &\text{Flow Delay} \quad d(t) = \inf\{\tau \ge 0 : R(t) \le R^*(t+\tau)\} \\ &\text{Curve Backlog} \quad b(t) = \alpha(t) - \beta(t) \\ &\text{Curve Delay} \quad d(t) = \inf\{\tau \ge 0 : \alpha(t) \le \beta(t+\tau)\} \end{aligned} \tag{6.4}$$

For all systems, an output arrival curve (that is, the arrival curve for the following node) can be calculated by deconvolution of the input arrival curve with the node's service curve. If no arrival curve is known for a

node, a minimal arrival curve can always be calculated by deconvolution of the input flow with itself. This leads to the following expressions:

$$\begin{aligned} &\text{Output Arrival Curve} \quad \alpha^* = \alpha \oslash \beta \\ &\text{Minimal Arrival Curve} \quad \alpha_{min} = R \oslash R \end{aligned} \tag{6.5}$$

Concatenation Whenever a flow passes through multiple nodes in sequence, it is possible to concatenate service curves into a single equivalent node. This is similar to concatenation of transfer functions in system theory. The service curve of the equivalent node is the convolution of passed service curves:

$$\text{Service Concatenation} \quad R^* \geq r_1 \otimes \beta_2 \geq (r_1 \otimes \beta_1) \otimes \beta_2 = R \otimes (\beta_1 \otimes \beta_2) \tag{6.6}$$

In NC, there is selection of basic curve functions that are frequently encountered when modelling networks. More complex curves can be constructed from basic functions by adopting a piecewise-linear approach (figure 6.4). A compilation of common basic functions in the context of NC is found in figure 6.5.

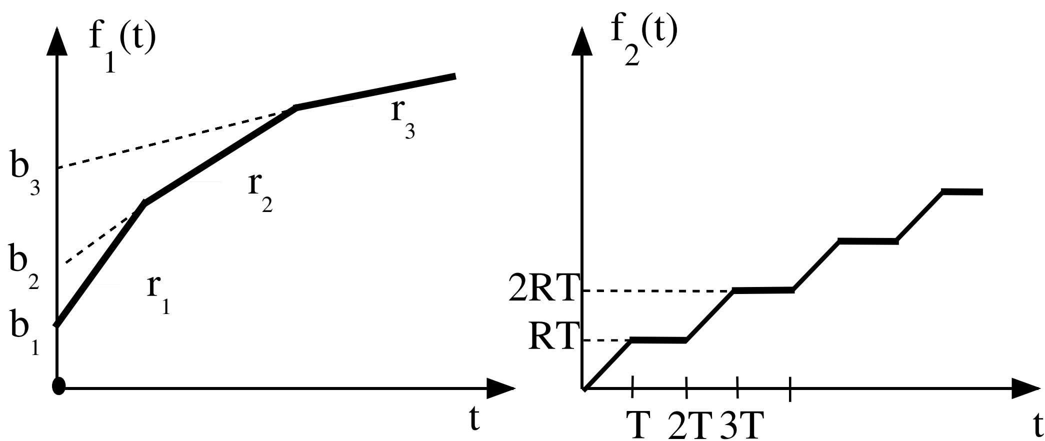
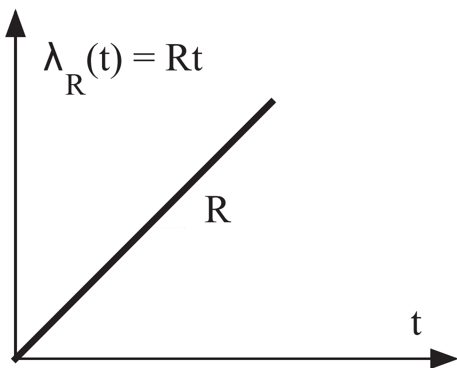
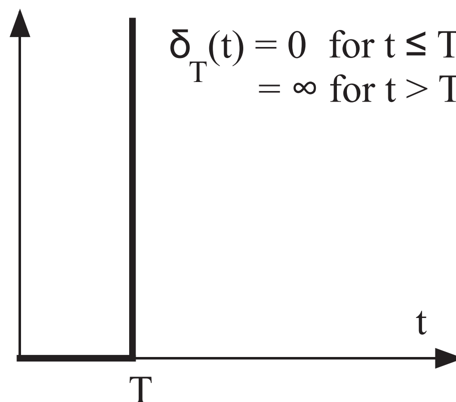


Figure 6.4: Examples of piecewise-linear Functions **thiran_network_2001**

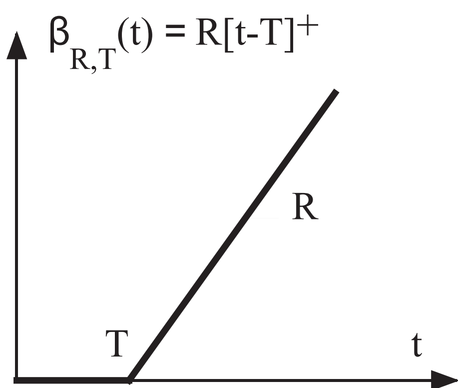
Peak rate function



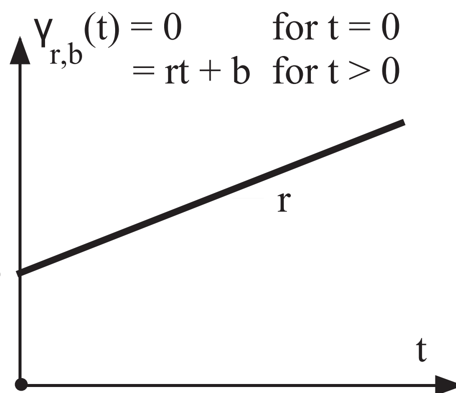
Burst-delay function



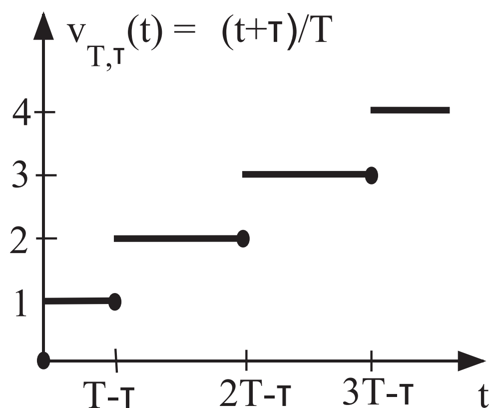
Rate-latency function



Affine function



Staircase function



Step function

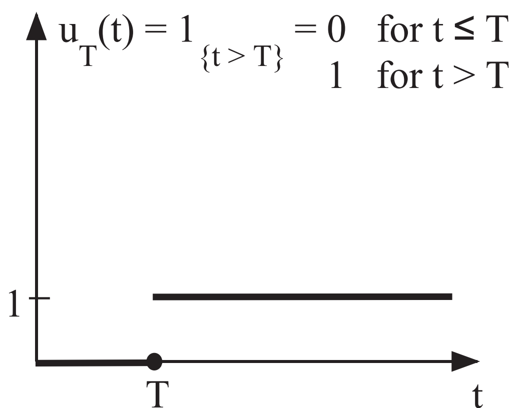


Figure 6.5: Catalogue of commonly used Curve Functions in NC **thiran_network_2001**

6.2.4 Elementary Building Blocks

NC uses a number of basic elements to construct network models, similar to system theory having different types of filter, mixer and splitter nodes as basic elements. The behaviour of a complex, composite system is then derived from the behaviour of these basic elements.

Shaper A shaper is an element offering its shaping curve as both a minimum and maximum service. As figure 6.6 shows, the convolution applies the shaping curve simultaneously at *every* point of the input flow, thus forcing the flow to have σ as its arrival curve. Whenever it would exceed the curve, data is delayed in time (moved to the right). There are two types of shapers defined in **thiran_network_2001**. The first complies to the above definition, although it is left unclear how the process is actually implemented. Focusing on the second definition, elements called *greedy shapers*. A greedy shaper "*delays the input bits in a buffer, whenever sending a bit would violate the constraint* σ*, but outputs them as soon as possible*" **thiran_network_2001**.

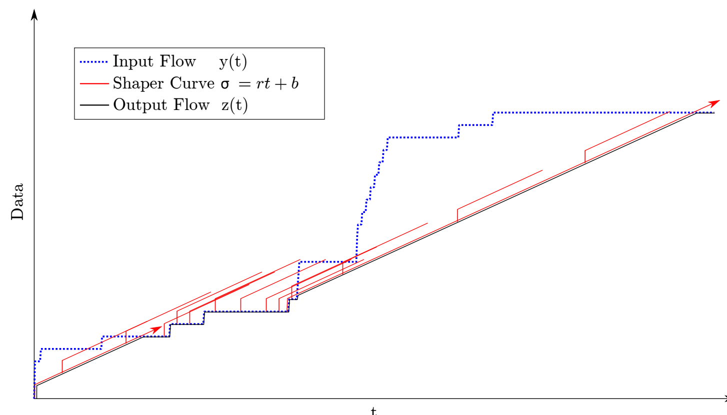


Figure 6.6: Visualisation of the Effect of min-plus Convolution:
Shaping curve σ is enforced at every point of input flow $y(t)$

Packetiser A packetiser is a variable delay element with a service curve roughly resembling a staircase (figure 6.7). It delays the output so it stays at step level $L(n-1)$ until the input flow has reached $L(n)$. Packet data is thus only forwarded once the whole packet was received **thiran_network_2001**. An element behaving in the way described is the so called "L-Packetiser". These are theoretical constructs, assuming instantaneous packet arrival

and departure. It employs the indicator function $1_{\{<expr.>\}}$, which is defined as

$$1_{\{<\text{expr.}>\}} = \begin{cases} 1 & \text{when } \textit{expression} \text{ is true} \\ 0 & \text{when } \textit{expression} \text{ is false} \end{cases} \tag{6.7}$$

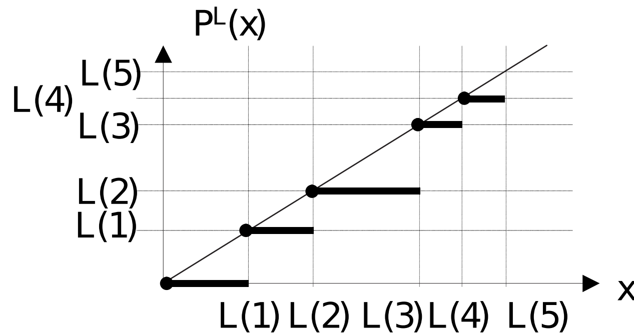


Figure 6.7: Definition of Function P^L **thiran_network_2001**

The function for an L-Packetiser can be then written as eq. 6.8, x being the current flow level ($R(t)$).

$$\mathrm{P^L(x)} = \sup_{n \in \mathbb{N}} \left\{ L(n) 1_{\{L(n) \leq x\}} \right\} \tag{6.8}$$

We will now take a closer look at the step function $L(n)$ which the packetiser employs to delay data until a packet is complete. $L(n)$ is the cumulative function of packet lengths, $l(n)$ being the length of the n-th packet. $L(n)$ is defined as

$$\begin{aligned} \mathrm{L(0)} &= 0 \\ \mathrm{l(n)} &= L(n) - L(n-1) \\ \mathrm{l_{max}} &= \sup\{l(n)\} \end{aligned} \tag{6.9}$$

For variable length packets, step values for $L(n)$ are either delivered a-priori or calculated iteratively from $L(n-1)$. If packet length is constant, cumulative packet length can simply be written as

$$\mathrm{L(n)} = n \cdot l \tag{6.10}$$

L-packetisers are a virtual construct, because no components can provide instantaneous arrival *and* departure (except wires if relative times are considered). A more realistic application of the theory are Packetised Greedy

Shaper (PGS), which, as the name suggests, model buffer delay by prefixing the L-packetiser with a greedy shaper:

$$\mathrm{R}^* = P^L(R \otimes \sigma) \tag{6.11}$$

An L-Packetiser buffers a packet, the first bit of any incoming packet is delayed until the arrival of its last bit. Assuming a constant rate shaper λ_C as the bit-by-bit system, the maximum delay experienced by a packet is the time it takes the maximum size packet, $\frac{l_{max}}{C}$. The equivalent minimum service of a packetiser can therefore be written as the concatenation of a constant rate node and the buffering delay, a rate-latency service of the form $\beta_{C,\frac{l_{max}}{C}}$.

Multiplexer Multiplexers are the most complex building elements in NC, because their impact can vary widely depending on their inputs and policy. Aggregation of flows is a common scenario for routers, switches or even endpoints, if they run multiple services in parallel. The multiplexer is a node offering a total service β, which is usually a constant rate of λ_C, to all incoming flows $\sum R_i(t)$. Service allocation to the individual flows is defined by a scheduling policy. The main distinction is made between arbitrary multiplexing, which assumes no knowledge about policy, and several other cases. The assumption of an arbitrary policy always provides correct, but usually most pessimistic bounds. The most important other case is First In, First Out (FIFO) multiplexing, which assumes messages are served in order of their arrival. FIFO and fixed priority policies (one input flow always preferred over another) are also handling well in NC.

We shall give an example of a simple fixed priority setup with two flows. The service available to the Low Priority (LP) flow is the residual service, after the preferred flow has been served. The High Priority (HP) flow's arrival curve is subtracted from the available service, which amounts to the residual service for the LP flow. Because service curves are always wide-sense increasing **thiran_network_2001**, so the supremum of the difference must be used **schmitt_comprehensive_2007**.

Whenever the slope of the arrival curve is greater than the slope of the service curve, the difference would be a decreasing function. This is in effect service backlog, but service curves cannot represent this directly. The delay this backlog causes is added instead. This is achieved through the supremum, as it keeps the residual service at constant level when it would be decreasing. Since no arrivals are serviced during that period, it introduces an equivalent delay. The shorthand notation for this residual

service is defined as

$$\beta_2^{l.o.} = \sup_{0 \le s \le t} \{\beta(s) - \alpha_1(s)\} = \beta \ominus \alpha_1 \tag{6.12}$$

More explicitly, consider a multiplexer offering a constant service rate C. The high priority flow is constrained by an affine function of maximum rate r and a buffer size b, with $r < C$. It is easy to see that the rate experienced by the LP flow will be the difference between the HP rate and the available rate in the system. Furthermore, if the HP flow has backlogged traffic, it can completely saturate the system. This makes the LP flow wait until all HP backlog is serviced. Thus the residual service is a rate latency curve

$$\beta^{l.o.} = \beta_C \ominus \alpha_{r,b} = \beta_{C-r, \frac{b}{C-r}} \tag{6.13}$$

If there is more than one flow to be multiplexed, the residual service experienced by the flow of interest is calculated by summing up the arrival curves of all interfering flows, such that

$$\beta_{foi}^{l.o.} = \beta \ominus \sum_i \alpha_i \tag{6.14}$$

Depending on the system's arbitration policy, a peculiarity can occur for HP service. If the system operates under a FIFO policy, HP itself has a waiting condition, due to an ongoing LP transmission (because it cannot pre-empt the LP flow). In line with FIFO, there must exist an upper bound for the size of such units of data, l_{max}. Because the length is arbitrary, this holds for bits and whole packets alike. HP is waiting for any ongoing LP transmissions to complete, so its minimum service is defined as

$$\beta_1^{hi} = \beta \ominus l_{max} \tag{6.15}$$

Scaler In many networks, there are nodes that compress or decompress data (video-encoder, etc.). This is a problem in NC, because the fundamental criterion, that $R(t) \le R^*(t)$, is violated in the case of compression. While it is kept in the case of decompression, the relation between input and output flow is still distorted. In both cases, horizontal and vertical deviation no longer correspond to delay and backlog a flow is experiencing.

Data Scaling, first introduced by **fidler_way_2006**, is a concept for NC which handles this problem by application of scaling curves **fidler_way_2006**. The Scaler will assign each bit of data $a = R(t)$ a scaled image of $S(a)$. S is a wide-sense increasing, bijective function curve, thus an inverse function

S^{-1} exists. From the perspective of the system's ingress, delay and backlog can then be calculated since perspective is important because R_S is scaled in relation to the ingress, but not in relation to a downstream node.

$$
\begin{aligned}
\mathrm{R_S^*(t)} &= S(R(t) \otimes \beta(t)) \\
\\
\mathrm{b(t)} &= R(t) - S^{-1}(R_S^*(t)) \\
\mathrm{d(t)} &= \inf\left\{\tau \geq 0 : R(t) \leq S^{-1}(R_S^*(t+\tau))\right\}
\end{aligned}
\tag{6.16}
$$

Scaling Functions and Curves Scaling functions can be directly applied to flows. However, application to arrival and service curves is not directly possible. Scaling curves must be derived from the function first. $\underline{S}$ is a minimum scaling curve of S if it is less or equal to its max-plus deconvolution, likewise, $\overline{S}$ is a maximum scaling curve of S if it is less or equal to its min-plus deconvolution:

$$
\begin{aligned}
\underline{\mathrm{S}}(\mathrm{b}) &\leq \inf_{a\in[0,\infty)} \{S(b+a) - S(a)\} &= S(b) \,\overline{\oslash}\, S(b) \\
\overline{\mathrm{S}}(\mathrm{b}) &\leq \sup_{a\in[0,\infty)} \{S(b+a) - S(a)\} &= S(b) \oslash S(b)
\end{aligned}
\tag{6.17}
$$

Inverse Scaling Curves Obtaining inverse scaling curves follows directly from eq. 6.17 by applying the process to the inverse scaling function S^{-1}. It further holds that the maximum scaling curve of the inverse scaling function, $\overline{S^{-1}}$, is equal to the inverse of the minimum scaling curve of the scaling function, $\underline{S}^{-1}$. and vice versa **fidler_way_2006**.

Scaled Servers To apply the above statements to finding an end-to-end delay bound, it is necessary to scale servers. This means applying scaling curves to service curves to obtain a scaled service. This may seem trivial, but is important as a way to allow concatenation of systems in the presence of scalers.

The core concept is the equivalency of the following systems: The minimum and maximum service, β and γ, of a server with scaled output (a) and a server with scaled input (b) lead to equivalent bounds, if S is bijec-

tive and:

$$\begin{aligned} \beta^{\mathrm{S}}(\mathrm{t}) &= \underline{S}(\beta(t)) \\ \gamma^{\mathrm{S}}(\mathrm{t}) &= \overline{S}(\gamma(t)) \\ \\ \beta(\mathrm{t}) &= \underline{S^{-1}}(\beta^{S}(t)) \\ \gamma(\mathrm{t}) &= \overline{S^{-1}}(\gamma^{S}(t)) \end{aligned} \tag{6.18}$$

6.2.5 Delay Analysis Methodology

There are many different approaches to obtain end-to-end delay bounds for a system. The first three, *Total Flow Analysis (TFA)*, *Separate Flow Analysis (SFA)* and *Pay Multiplexing Only Once (Analysis) (PMOO)*, are the "classical" approaches, solely relying on network calculus. The different methods are demonstrated on a minimal example of two nodes in tandem **schmitt_comprehensive_** through which two flows are multiplexed as a FIFO aggregate.

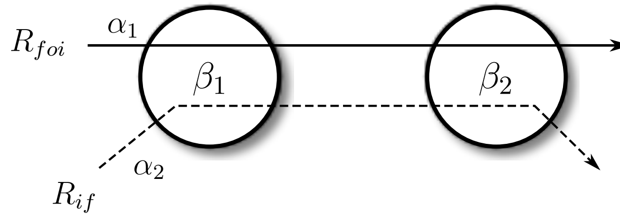


Figure 6.8: Minimal Network Example: 2 Nodes, 2 Flows

Total Flow Analysis This form of an end-to-end analysis adds the delays encountered by the total flow, that is, the *sum* of all flows, along the path. This is calculated per node, using the arrival curves transformed by the node. This means using the original arrival curves α_1 and α_2 at the first node, their output arrival curves α_1' and α_2' (see eq. 6.5, p. 61) at the second node and so on. Delay is defined as horizontal deviation between input and output flow, so the delay for TFA is calculated as

$$d_{TFA} = h(\alpha_1 + \alpha_2, \beta_1) + h\big((\alpha_1 + \alpha_2) \oslash \beta_1, \beta_2\big) \tag{6.19}$$

The obtained bound is valid for both the Flow Of Interest (FOI) and the interfering flow(s), but does not provide information about which flow it belongs to. It is thus overly pessimistic for all but one flow. TFA tends to produce the least tight bounds.

Separate Flow Analysis SFA aims to obtain a tight bound for the FOI by removing it from the system and inspecting the residual service available to it after servicing the interfering flow(s). This is achieved by summing up all flows except the FOI and subtracting the aggregate flow from the total service of the system (see eq. 6.12, p. 66). The residual service is then computed over all nodes by convolution, which equals the end-to-end service encountered by the FOI and thus provides its delay. This can be written as

$$d_{SFA} = h\Big(\alpha_1, (\beta_1 \ominus \alpha_2) \otimes (\beta_2 \ominus (\alpha_2 \oslash \beta_1))\Big) \tag{6.20}$$

Because SFA includes topology information when calculating residual service, it is proven to deliver tight bounds for all multiplexing policies. Contrary to TFA, SFA pays bursts only once (PBOO criterion), because residual services are concatenated before calculating end-to-end delay.

Pay Multiplexing Only Once Analysis The downside of SFA is an overly pessimistic accumulation of multiplexing delay at every node, even if it is not occurring there. Consider two FIFO multiplexed flows, which can both send at the same rate, passing through several nodes all capable of this rate. While it is true that only one flow can send at the same time, this only determines delay at the first node. Once the order of sending is determined, additional nodes should not introduce more multiplexing delay.

PMOO is a special case of SFA which tries to, as the name suggests, avoid paying multiplexing multiple times by concatenation all encountered nodes into one equivalent node before the residual service calculation. In the example from figure 6.8, this means the convolution of β_1 and β_2 before subtracting α_2:

$$d_{PMOO} = h\Big(\alpha_1, ((\beta_1 \otimes \beta_2) \ominus \alpha_2)\Big) \tag{6.21}$$

In most cases, PMOO delivers the tightest bounds. There are some special cases when SFA can perform better than PMOO, when the service rates is higher at downstream nodes than at the ingress **schmitt_delay_2008**. Contrary to SFA, PMOO analysis has only been proven for arbitrary multiplexing policy.

Building on NC There are known problems with the described analysis methods when treating aggregated flows, because it can be proven **schmitt_delay_2008** that even with PMOO analysis, multiplexing over multiple hops does not

always produce the tight bounds. There were more recent advance in providing tight end-to-end delay bounds from Lencini et al. **lenzini_end–end_2007** and Schmitt et al. **schmitt_delay_2008**(2008), employing optimisation algorithms on top of network calculus. In order to obtain the minimum delay bound, these approaches define the service curve of all traversed nodes and then solve a linear optimisation problem for the distribution of backlog between these.

The most recent research by **bondorf_delay_2016** from 2016 shows a very interesting development back to pure algebraic solutions. In their publication, they prove the existence of a completely algebraic technique, which requires much less computation than linear optimisation, but still closely matches experimental results to within 1.16% **bondorf_delay_2016**.

6.3 Approach for modelling the Data Master

6.3.1 Overview

The following is the overview of the intended NC model to be used for an end-to-end delay analysis of the DM and its environment. The first part of the model will be the DM itself and its sub-components. The second part will be a black box model of the WR switches, the third a model of TR. While it would be feasible to model the switches more accurately, a white box approach is outside the scope of this work.

The purpose of this model (and indeed of NC as a whole) is to provide worst case bounds on delay, backlog and output flows, it does not provide exact input-output transformations. While it is technically possible to create accurate transfer functions, the benefit of using "low-level" models would be limited to verifying the formally proven abstract modelling techniques provided by **thiran_network_2001** and Schmitt.

Naming Conventions Within the timing system, server nodes are processing data at four different (three distinct) rates r_x. These are, in descending order:

$$\begin{aligned} r_3 &= 4\,\mathrm{B} \cdot 8\,\mathrm{ns}^{-1} &&= 4\,\mathrm{Gbit\,s}^{-1} \\ r_{2a} &= 4\,\mathrm{B} \cdot 16\,\mathrm{ns}^{-1} &&= 2\,\mathrm{Gbit\,s}^{-1} \\ r_{2b} &= 2\,\mathrm{B} \cdot 8\,\mathrm{ns}^{-1} &&= 2\,\mathrm{Gbit\,s}^{-1} \\ r_1 &= 1\,\mathrm{B} \cdot 8\,\mathrm{ns}^{-1} &&= 1\,\mathrm{Gbit\,s}^{-1} \end{aligned}$$

6.3.2 Machine Schedules as Flows

Before constructing a detailed service model, we shall have a closer look at the input flows, i.e. how they originate from a collection of machine schedules. All flows entering from the CPU side are of potential interest for analysis, while the injected headers from the Etherbone Master (EBM) and WR traffic will always be treated as interfering flows. Machine Schedules provide content for timing messages as well as information about points of decision, i.e., which schedules can be played next and which is the default selection. Each message within a schedule requires a dispatch time in relation to its time offset. With these offsets and their arrival time, an Earliest Deadline First (EDF) scheduler can create the corresponding message flow, the cumulative sum of messages over time.

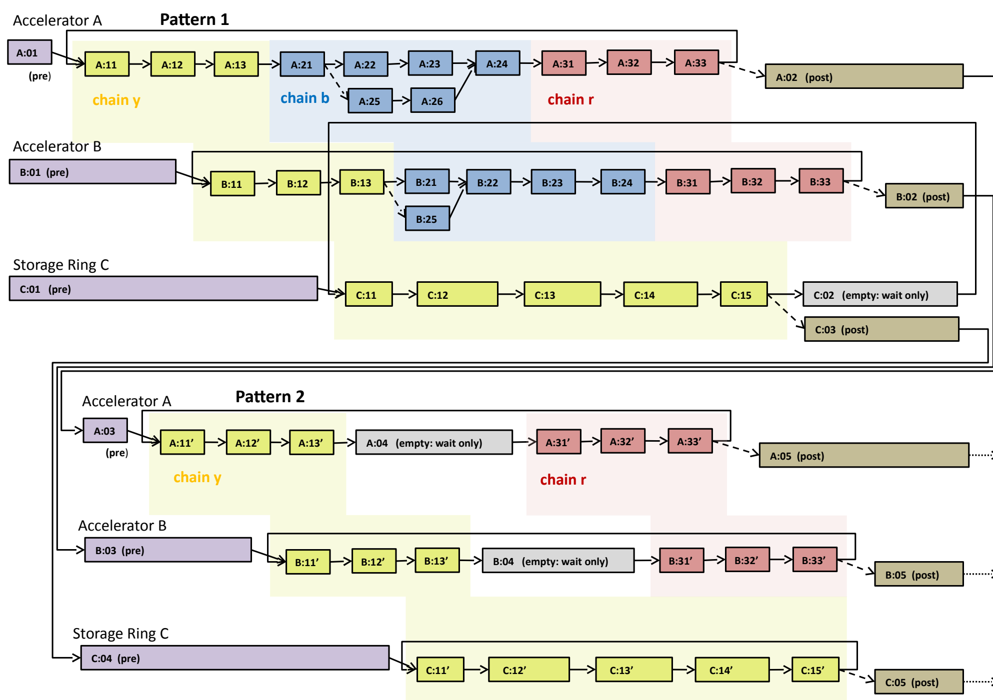


Figure ??: Machine Schedules for the Accelerator (p. ??)

Assignment of Arrival Curves It is always possible to find a minimal arrival curve for a flow. This is obtained by the min-plus deconvolution of the flow with itself, thus $\alpha = R \oslash R$. These arrival curves tend to be *very* form-fitting to the actual flow and are therefore tedious to describe

formally. They are also often not concave, but star-shaped. For convenience (and especially for the following worst case of several schedules), it is advantages to find an approximated arrival curve from within a certain family of functions. A piece-wise affine function as shown in figure 6.4 tends to provide a good approximation **thiran_network_2001**. We will only briefly touch the subject of a suitable approach for approximation of arrival curves, as a full investigation is out of the scope of this work.

The solution to the problem is finding the minimax solution, that is, minimisation of the maximum error when choosing affine segments. Apart from least square approximation, which does not necessarily converge **imamoto_recursive_200** there are splitting algorithms that search for segments within a certain error criterion **vandewalle_calculation_1975** and also recursive approaches, which try to find the location of the tangent pivots directly **imamoto_recursive_2008**. While **imamoto_recursive_2008** will find an optimal solution, the result is only proven to be optimal for concave or convex functions. This is a problem for minimal arrival curves: While concave functions are always sub-additive, sub-additivity does not imply concavity. It would therefore be necessary to either construct the concave hull of the minimal arrival curve before applying **imamoto_recursive_2008**, evaluate the quality of fits to sub-additive functions or choose for example the algorithm of **vandewalle_calculation_1975 vandewalle_calculation_1975**, which is applicable to arbitrary functions.

It is important to note that assignment of arrival curves needs to consider the *whole* flow, i.e. the concatenation of machine schedules. All successions of machine schedules in the DM are either finite or periodic for the validity period of the analysis. Once a fitting arrival curve has been defined, it is possible to assign service to this flow at every node it passes through. This will allow to obtain an end-to-end delay bound for the corresponding flow.

As a proof of concept for this approach, an example for the generation of a piece-wise affine arrival curve describing a periodic message flow is given in the following paragraph.

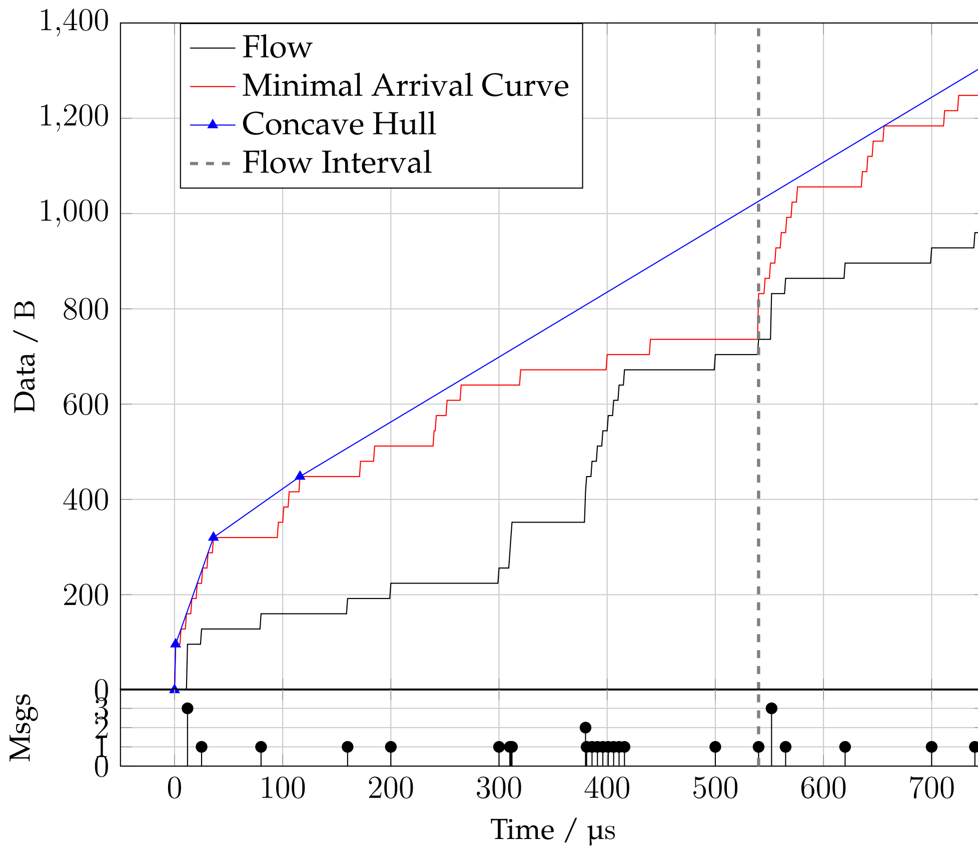


Figure 6.10: Generation of a piece-wise affine Arrival Curve from Flow. The corresponding Messages are shown in the stem plot below.

The example given in figure 6.10 constructs the affine arrival curve for a periodic flow. The corresponding message flow is visualised as a stem plot at the bottom of the figure. Each circle signifies as a timing message of $32\,\mathrm{B}$, stacked circles show concurrent execution times. As a first step, the flow's vector is cloned and concatenated to the original (black curve). Secondly, the minimal arrival curve is calculated by min-plus deconvolution of the flow with itself (red curve). Because the flow was cloned before, the minimal arrival curve (within the flow's interval) matches a periodic repetition. As the third step, the concave hull of the minimal arrival curve is constructed (blue dashed curve). All nodes not contributing to the outer shape are removed. As the fourth step, the affine function is reduced to the length of the flow's period, keeping the slope it had at the end of the interval. The resulting function is piece-wise described by affine functions of the form $m \cdot x + b$. It has a peak rate of $\approx 5\,\mathrm{Mbit\,s^{-1}}$ and a sustainable rate of $\approx 1\,\mathrm{Mbit\,s^{-1}}$, with an initial burst of $96\,\mathrm{B}$. Eq. 6.22 shows the description

of this arrival curve (burst values in bytes, rates bytes per second, time in micro seconds):

$$\alpha_{r_i,b_i} = \left\{ \begin{array}{lrl} \gamma_{96,\;\;6.5\cdot 10^5} & 0 & < x < \;\; 360 \\ \gamma_{320,1.28\cdot 10^5} & 36 & \leq x < 116 \\ \gamma_{448,1.25\cdot 10^5} & 116 & \leq x < \infty^+ \end{array} \right\} \tag{6.22}$$

6.3.3 Outside Interference

Worst Case for Online Flow Control Outside intervention through interlocks will change the path through the machine schedule graph (see figure **??**) in realtime, yet the delay analysis will be done offline. The reason is that even if the analysis is carried out in realtime, detecting an imminent violation of the system's delay bound will not help containing the situation. Instead, the worst case combinatorial scenario of all possible flows at this point must be considered by taking the supremum of all minimal arrival curves. The supremum of sub-additive curves is also sub-additive, preserving their property.

- Arrival curves of all involved Schedules
- Time of Points of decision
- Sets of alternative arrival curves for each Point of Decision

If this information is available, it is possible to create compound worst case arrival curves from several individual ones by supremum of all alternatives. In the case of the DM, the combinatorial arrangement can only be conducted *after* the first EDF scheduler in the Lattice Mico 32 Processor (LM32)'s firmware. The reason is finding the worst case combination is only possibly with arrival curves of flows in which messages already occupy their scheduled release times.

Relaxed case for optional Online Flow Control All requests from experiments are not regarded as time-critical, they can thus be delayed without penalty. This creates a degree of freedom in online flow control, as each requested change to the schedule configuration, can instead be included by re-computation of the delay analysis. If the delay bound is violated, the change will not be executed and there are several possibilities to solve the problem. These range from telling the operator that this change is not allowed to automatically shifting the desired schedule change in time until the system can provide a suitable service. Arrival curves thus do not need

to cover all possible combinations of requests from experiments, only the ones currently selected.

6.3.4 Recurring Analyses

At the time the very first analysis is undertaken, the model is representing a system at time $t = 0$ and thus without history. When the analysis is repeated during runtime on change of machine schedules, it is obvious that the system is *not* in this state. There are three possible approaches to treating case study, the first being trivial:

Clean Slate The first possibility is to halt the system completely. Because all messages were scheduled to spend a maximum time Δ_t in the system, ceasing the input flows and waiting for Δ_t will guarantee a system with empty buffers, hence the system is equivalent to the state at $t = 0$. This will initially be the preferred mode of operation for the case study.

Prepare for Everything The second possibility is accepting more loose arrival curves for the input flows and cover *all* possible combinations of machine schedules per input flow, thus never needing a second analysis. This would work for smaller sets of schedules and can be complemented by rare resets as described in the first approach.

Time Stop The third option involves halting time at the point of change, calculating backlog at every node, update input flows according to the requested changes, apply **thiran_network_2001**'s theorem **thiran_network_2001** for shapers with non-empty buffers and update service curves.

6.4 Scheduler Models

Type The DM requires two layers of schedulers to sort timing messages into chronological order by their deadlines. All schedulers are implemented as packetised earliest deadline first based on delay values.

Lower Layer The lower level scheduler is implemented in hardware and aggregates the flows from all instantiated processors. This module has been dubbed Hardware Priority Queue (Priority Queue (PQ)) and has been described in detail in chapter **??**.

Upper Layer The upper level of schedulers is implemented in firmware inside the processors, as presented in chapter **??**. The scheduler would not strictly be necessary at this point, but does allow a better utilisation of available processing time. Machine schedules must be distributed to individual CPUs depending on their current utilisation, as it must always be $\leq 100\%$. While it is possible to aggregate all these schedules into one big schedule per processor before running, this would be very inflexible. Every update would require a complete stop and exchange of the whole aggregate. Instead, software EDFs can easily aggregate individual machine schedules, each assigned to a worker thread, into a chronological flow. Such a scheduler is thus a prerequisite to enable online update of machine schedules and online flow control as described in **??** and 6.3.3.

6.4.1 Scheduling under Network Calculus

Both schedulers are treated here within the DM as delay based schedulers, which means the decision for the next packet to service is done by the remaining delay budget per packet. A delay budget is spanning the time from a packe's arrival to its latest possible departure. The general schedulability criterion is derived from the maximum horizontal deviation between the sum of all arrival curves and the available service $\beta = \lambda_C$. If it is finite and not greater than the maximum allowed delay budget, the set of arrival curves is schedulable.

$$\sum_i \alpha_i(t - d_i) \leq \beta \tag{6.23}$$

In the case of the DM, this presents a problem, as each packet's delay budget is referring to its execution at the endpoint, not the departure time at that particular DM node. Each local deadline is a part of the total delay budget, but it is unknown. A rough estimate can be given once all static delays are known and subtracted. If all delays from cross traffic are bounded as well, exact calculation is possible, but this does not provide any additional information at this point. The scheduler equation can be used, however, as a simple and fast instrument to detect overload by a set of flows before attempting a full delay analysis.

6.4.2 Soft-CPU Scheduler

The input flows in CPUs are derived from timing messages in machine schedules. It is the purpose of the CPU scheduler to chronologically sort

and aggregate messages from all threads and send them as early as possible within the time window of $D_j - \Delta_t$.

Joining Two Worlds The very first point to address is the existence of several distinct domains within the DM: CPU, Wishbone (WB) bus and Network. The latter two relate and are thus trivial to describe in NC, as they only differ in bandwidth and in the network always being packetised while WB is a cycle based bus. The relation between programs in a DM CPU and bus/network activity is not trivial though and we shall start modelling the DM with an approach for a CPU/traffic relation.

CPU Activity vs Generated Traffic An LM32 CPU can be described as an NC node, offering a constant rate service (operations executed over time), and a program as a flow (cumulative operations over time). The output flow (of interest) is all bus activity downstream towards the network interface. This means that a program is already an aggregate of flows. They are flows that generate traffic downstream and flows that do not, i.e. message and overhead flows. The composition of overhead and its impact on message service are discussed here.

Overhead Concept Consider the following: The processor arbitrates its computing power between a number of N threads and the scheduler itself. According to chapter **??**, it is assumed that all tasks have a deterministic execution time which is previously known. The scheduler itself also has a deterministic execution time. So there is not only a CPU rate, but also a message rate, the maximum rate the firmware can send messages at. While formal investigation of program execution time can be a highly complex and computationally intensive task, measuring the maximum message rate using the CPU cycle counter or a logic analyser is trivial.

The scheduler further has a dispatch function f, which transforms a skeleton message from RAM into a timing message on the bus. Let there be another function g which sends synchronisation messages to CPUs. The function g does not produce messages on the timing network, and since it uses the Message Signalled Interrupt (MSI) WB bus, it has no impact on the normal WB traffic neither. Because the effort of preparing and sending synchronisation and timing messages is very similar, f and g can be assumed to have equal execution times.

Sync messages are part of machine schedules, all message flows therefore have an associated sync overhead flow. This produces the interfering

flows in the CPU node. Sync flows are of no further interest to the analysis, as they are extracted again directly after the CPU node. Only their effect on the CPU's residual service curve to the messages is considered.

Overhead Flows Figure 6.11 shows the block diagram of the CPU service node, a constant rate server used by the message flows and several interfering overhead flows.

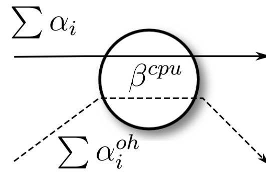


Figure 6.11: CPU Scheduler node

The leftover service available to all timing messages is the total service of the CPU after per flow sync overhead α_{oh_i} has been subtracted:

$$\beta^{\text{cpu}} = \lambda_{r_3} \ominus \sum_i \alpha_i^{oh} \tag{6.24}$$

Actual Implementation We know there exists a limit Δ_t dictated by the control loop speed, which is the end-to-end delay budget of a message. In the present case, it signifies the minimum time before its deadline D_j a message shall be dispatched. The actual implementation determines the task with the minimum deadline at the very moment its predecessor was serviced. Afterwards a separate check is run periodically if this message is eligible for dispatch, that is, if $t \geq D_j - \Delta_t$. If it is positive, the message is sent. The rate of this check is the same as the service rate offered to messages by the CPU.

Simplification Representation in NC can be simplified by reordering these steps and crafting slightly different input flows. Instead of letting new messages arrive immediately after service, messages can be placed in accordance with their arrival times $D_j - \Delta_t$. This already takes care of the eligibility window Δ_t. Feeding such a flow through the CPU's service window will then introduce the same delay as in the prior case.

CPU Schedulability It is first necessary to obtain information about the possible size of the delay budget d for the scheduler input flows. However,

it is not possible to determine the budget in the present case. The delay budget for EDF schedulers is defined as the maximum time between arrival and departure *at the server containing the scheduler*. In the present case, this partial budget is unknown - only the end-to-end budget is. A loose approximation can be obtained by deducting the sum of all static delays $\sum \delta$ (which will be deduced in this chapter) from the end-to-end budget Δ_t. Note that being schedulable is no guarantee for timely arrival with regard to the endpoint, but unschedulable flows are guaranteed to be late. A general schedulability criterion for each processor node is:

$$\sum_i \alpha_i(t-d) \leq \beta^{cpu}(t) \qquad \text{with } d = \Delta_t - \sum \delta \tag{6.25}$$

6.4.3 Processor Output

Since β^{cpu} is known, the output flow of a CPU can be derived. For TFA, the output flow and arrival curve can be calculated with the aid of the sum of all inputs:

$$R^* = \sum_i R_i \quad \otimes \beta^{cpu} \qquad \alpha^* = \sum_i \alpha_i \quad \oslash \beta^{cpu} \tag{6.26}$$

The leftover service curve required for both SFA and TFA on the other hand can be obtained from a slight variant of eq. 6.24. The overhead caused by the scheduler, all sync flows and all interfering message flows can be subtracted from the CPU's service, resulting in the residual service for the message flow of interest:

$$\beta_{\text{foi}}^{\text{l.o.}} = \lambda_{r3} \ominus \left(\sum_i \alpha_i^{oh} + \sum_{i \neq foi} \alpha_i \right) \tag{6.27}$$

6.4.4 Priority Queue Scheduler

We now have the processors' outputs, which are packet flows with wide-sense increasing deadlines. The next node on the path is the PQ, the sec-

ond layer of EDF schedulers in the DM. Its purpose is the chronological aggregation of all input flows into one output flow, ordered by deadlines.

PQ Schedulability The constraint is that there must be no back-pressure to the CPU, so overflow of the input queues is not permitted. This is necessary to keep program execution deterministic and hence maintain the schedulability criterion. We will first derive an upper limit to the delay budget d_i from the buffer capacity of the input queue. The input queue can take in data at a rate of r_3, so it offers a maximum service γ_{r_3}. The delay d_i must therefore be the time it takes an incoming flow constrained by α_i (which must be sub-additive) to fill an input queue of size b. Since γ_{r_3} poses an upper limit on the input rate, we will convolute it with α_i, the min-plus convolution of two sub-additive functions being the minimum **thiran_network_2001** of both. This will result in the following schedulability criterion:

$$\sum_i \alpha_i(t - d_i) \leq \beta(t) \qquad \text{with } d_i = min\{s > 0 : min(\alpha_i, \gamma_{r_3})(s) = b\} \tag{6.28}$$

Abstract Model - Service Curves In the analysis of the abstract model of an EDF, the maximum delay depends solely on the total flow passing through the scheduler's constant rate node. In this case, a service curve containing a fixed delay T_a for evaluation of each timestamp and the impact of the packetiser in each queue ($\beta_{r3,\frac{l_p}{r3}}$) must be included, as all inputs experience this. This can be written as

$$\beta^{ch} = \delta_{T_a} \otimes \beta_{r3,\frac{l_p}{r3}} \tag{6.29}$$

The complete EDF scheduler in the PQ module features M channels, each connected to the constant rate node of the PQ.

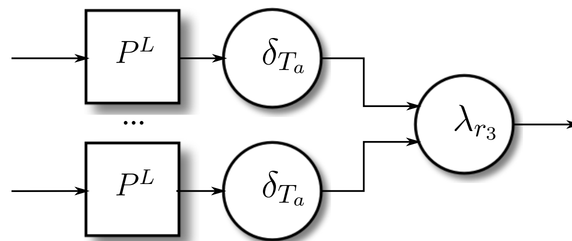


Figure 6.12: PQ Scheduler node

And with equation 6.29, we can finally calculate the residual service a single flow (at the PQ, which already can be aggregates) would experience:

$$\beta_{foi}^{l.o.} = \lambda_{r_3} \ominus (\sum_{j \neq foi} (\alpha_j \oslash \beta^{ch}) \qquad (6.30)$$

6.5 Etherbone Master – Framer

The EBM is responsible for wrapping WB accesses to other systems in the EB protocol, it creates a network packet and hands it over to the network interface.

6.5.1 Etherbone Master Functional Recap

The EBM gathers Wishbone Bus Operations and a framer sub-module analyses type, order and destination. It then generates appropriate EB record headers and inserts them as required. Once dispatch of the opened packet is requested, the EBM finalises and inserts the network header information, and starts transmission. More details can be found in chapter **??**. Because the EBM is controlled by the PQ in the present case, requests for dispatch can be caused by reaching the size limit or by hitting a timeout.

Following the path from the PQ downstream, we will begin by modelling the framer sub-module of the EBM.

6.5.2 Input Parser

The overhead produced by the framer depends on the WB operations. In the case of DM traffic though, only timing messages will arrive. These follow a fixed format of 8 consecutive write operations to the same destination, which makes the introduced record overhead constant (see chapter **??**).

Timing messages must not be split, and therefore the payload of EB records will be of a constant length l_p. This is packetisation at message level, and so the framer needs to contain a fixed length packetiser (see section 6.2.4) in the payload flow. Because l_p is constant, the length function $L(n)_p$ of the n-th payload packet is $L(n)_p = n \cdot l_p$. This results in the

following service curve for the message payload L-packetiser:

$$\mathrm{P}_{\mathrm{p}}^{\mathrm{L}}(\mathrm{x}) = \sup_{n \in \mathbb{N}} \left\{ l_p n \cdot 1_{\{l_p n \leq x\}} \right\} \tag{6.31}$$

The framer now needs a shaper prefixing the L-packetiser, which would be a guaranteed rate node of some arbitrary rate λ_r. The time T_a the framer requires to analyse the input needs to be accounted for. Because the analyser is pipelined and the input format fixed, this delay can be expressed as a simple guaranteed delay node δ_{T_a}. The two nodes can be concatenated to form a rate-latency node as the shaping curve σ:

$$\sigma_{r,T_a} = rt + T_a \tag{6.32}$$

Impact It is known from **thiran_network_2001** that a packetiser offers a minimum service described by the rate of the bit-by-bit system and the maximum delay from packet buffering, $\beta_{r,\frac{l_{max}}{r}}$. The impact of payload's packetised greedy shaper on the system's service is:

$$\beta_p = \beta_{r,\frac{l_p}{r}} \otimes \delta_{T_a} \tag{6.33}$$

6.5.3 Header Generation

The framer must now prefix each timing message with an appropriate EB record header of constant length l_h, which will create EB records of length $l_h + l_p$. While the header themselves are of no particular interest to an analysis, they are necessary in terms of the system service they consume. There are two different strategies to discuss through which injection of overhead can be modelled.

Scaled Flow Approach Consider the knowledge about the expected input and output flows. The output is obtained by injection of data of size l_h every l_p in the input flow. Assuming header insertion would happen instantaneously, gives

$$R^*(t) = R(t) \cdot \frac{l_p + l_h}{l_p} \tag{6.34}$$

Eq. 6.34 shows header injection to be in fact data scaling, if a bijective relation between header and payload size exists. The overhead can be modelled by application of a scaling function (see "Scaler", p. 63) in the EBM

and the inverse function in the TR. The scaling curve and its inverse are:

$$S_R(a) = a \cdot \frac{l_p + l_h}{l_p}$$
$$S_R^{-1}(a) = a \cdot \frac{l_p}{l_p + l_h} \tag{6.35}$$

6.5.4 Output Flow and Service Curves

Scaling is the simple and accurate representation of constant header insertion into packets of constant size.

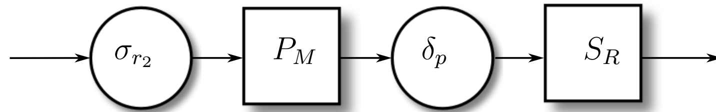


Figure 6.13: Block Diagram of EBM Framer Module

Figure 6.13 shows the resulting block diagram with packetiser, parser and scaler. With eq. 6.31, 6.32, 6.33, and 6.35, the output flow and service of the EBM framer can be modelled using the following equations:

$$R_f^* = S_R\big(P_p^L(\sigma \otimes R) \otimes \delta_{T_p}\big) \qquad \text{6.31, 6.32, 6.35} \tag{6.36}$$

$$\beta_f = \beta_{r_2, \frac{l_p}{r_2}} \otimes \delta_{T_p} \qquad \text{6.33, 6.35} \tag{6.37}$$

Scaling It is noteworthy that the service does *not* include the scaling function S_R, although the output flow R_f^* does. **fidler_way_2006** show **fidler_way_2006** that the order of scaling and service elements is interchangeable within a certain rule-set. In order to obtain a suitable equivalent system, it is necessary to consider the complete network, not individual modules.

All scaling effects spilling over to downstream modules are therefore ignored until reaching the analysis section 6.10. Instead only the scaling functions and unscaled service equations are provided.

6.6 Etherbone Master – TX

The purpose of the Transceiver (TX) sub-module is to take in a variable number of EB records and, by prefixing it with a packet header, turn them

into network packets. The maximum packet length and how long TX should gather EB records before producing a packet is configurable.

Header and Payload Size Let l_{nh} be the aggregated size of all headers for the IEEE 802.3 Ethernet (Eth), Internet Protocol Version 4 (IP), User Datagram Protocol (UDP) and EB protocol. Let l_p^S be the length of a timing message with an EB record header. The number of messages going into the same packet has a constant upper limit given by the maximum payload size, $l_{max} - l_{nh}$, and a constant lower limit of one message, l_p^S.

6.6.1 Variable Length Function

It is clear that waiting for a full packet is not an option, since the delay would be inversely proportional to the arriving flow. This is an undesired effect, as incoming flows would need to be padded to reduce delay and an equilibrium between transmission delay from the rest of the system and waiting time in the EBM TX would have to be found.

Timeout A timeout was introduced to bound the delay, which requires limiting the maximum waiting time for the first message to enter a packet. This results in a variable payload length $l_to(t)$. The payload packetiser therefore employs a length function $L(n)$, which provides cumulative packet lengths with a step-size being the minimum of l_{max} and the level at the timeout, l_{to}. Since $l(n) = L(n) - L(n-1)$, it is possible to calculate $L(n)$ from $L(n-1)$.

The timeout starts at the first message entering a packet, which is the case when $R'(t)$ (which is $R(t)$ after the packetisers bit-by-bit system) crosses the packet boundary (now $L(n-1)$). With $F(t)$ being the flow level at time t, we get

$$\mathrm{g(x)} = \inf_{s \in \mathbb{R}_+^*} \{s : F(s) > x\} \tag{6.38}$$

The timeout occurs after a timespan T. Unfortunately, this will introduce a problem: While R is packetised to be always multiple of l_p^S, it has to pass a bit-by-bit shaper, becoming $F = (R \otimes \sigma)(t) = R'$. Therefore, $R'(g(x) + T)$ can return a level right in the middle of a message (eq. 6.39).

$$\begin{aligned} &M = \{k \cdot l_p^S\}, \quad k \in \mathbb{N} \\ \\ &x \in M && \rightarrow && F\big(g(x)\big) \in M \\ &T \in \mathbb{R} && \rightarrow && \exists T : F\big(g(x) + T\big) \notin M \end{aligned} \tag{6.39}$$

To guarantee the delay bound, it is not possible to wait for a commenced message to fully arrive. To only put complete messages into a packet, it is necessary to round $R'(t)$ down to the nearest multiple of l_p^S. This means applying a floor function, which is equivalent to run $R'(t)$ again through the L-packetiser of the EBM framer:

$$\mathrm{R}''(\mathrm{t}) = \left\lfloor \frac{R'(t)}{l_p^S} \right\rfloor \cdot l_p^S = P_f^L\big((R \otimes \sigma)(t)\big) \tag{6.40}$$

And with eq. 6.40 and $F = R''$ so L-function for the network payload is:

$$\mathrm{L_{np}}(\mathrm{n}) = \min\left\{(L(n-1) + l_{max}), R''\big(g(L(n-1)) + T\big)\right\}, \qquad n \in \mathbb{N} \tag{6.41}$$

6.6.2 Header

The present case requires packets of variable payload length $l(n)$, yet with a header of fixed length l_{nh}. As mentioned in **fidler_way_2006**, scaling functions can be applied to the length function $L(n)$ of a packetiser. $L(n)$ is only point-wise defined for $n \in \mathbb{N}$ though, while scaling functions must be continuous.

$$\mathrm{P^L}(\mathrm{x}) = \sup_{n\in\mathbb{N}} \left\{L(n) 1_{\{L(n)\leq x\}}\right\} \tag{6.42}$$

However, the L-packetiser function from eq. 6.42 is defined for all real numbers **thiran_network_2001**, thus allowing the extension of $L(n)$ into the $\mathbb{R}_+^*$ domain by assigning each $a = R(t)$ a value from $L(n)$. So having obtained a continuous scaling function, scaling curves can be derived.

Scaling Function The L-packetiser equation is modified by a scaling function for constant header insertion. This means that each packet length $l(n)$ must be scaled by the addition of l_h, so that $S_P^p(l(n)) = l(n) + l_{nh}$. With the definition of the length function given by $L(n) = L(n-1) + l(n) \to L(n) = \sum_i^n l(i)$, the point-wise defined scaling function S_P^p now becomes:

$$S_P^p(L(n)) = \sum_i^n (l(i) + l_h) = L(n) + n \cdot l_{nh}$$

Combination with the L-packetiser equation 6.42 provides the scaling function S_P defined for $\mathbb{R}$, swapping the pre-factor for the indicator function

with its condition provides the inverse:

$$\mathrm{S_P}(\mathrm{a}) = \sup_{n \in \mathbb{N}} \left\{ (L(n) + n \cdot l_{nh}) 1_{\{L(n) \le a\}} \right\} \tag{6.43}$$

$$\mathrm{S_P^{-1}}(\mathrm{a}) = \sup_{n \in \mathbb{N}} \left\{ L(n) 1_{\{L(n) + n \cdot l_{nh} \le a\}} \right\} \tag{6.44}$$

The appropriate minimum and maximum scaling curves can once again be derived from max-plus and respectively min-plus deconvolution of the scaling function with itself.

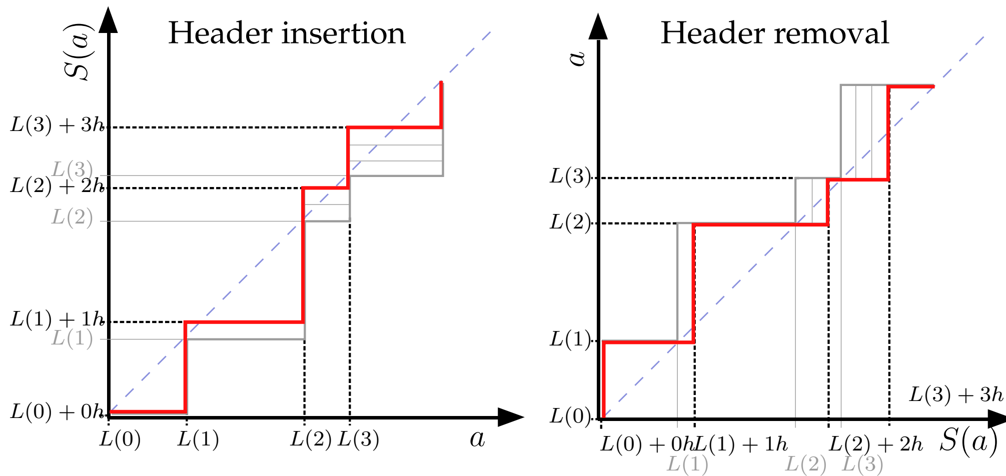


Figure 6.14: Packet Header Scaling Functions

Figure 6.14 illustrates the scaling functions (thick red line) for packet header handling. The left plot shows header insertion with eq. 6.43, the right the removal via the inverse function, eq. 6.44. The diagonal mirror axis is sketched in to demonstrate function inversion.

6.6.3 Finding Limits for Payload Length and Timeout

With the presented scaling curve, it is possible to calculate the exact introduced overhead at any given point in time for a specific flow. However, by careful choice of the payload limit and timeout value of the packetiser, it is possible to constrain the setting in a way that allows simplification of the scaling curve to a constant factor.

Impact of Payload Length The behaviour of the TX module is governed by two parameters, the allowed payload length Φ and the timeout T. Bandwidth utilisation depends on the overhead to payload ratio, the larger the

payload, the better. As Φ approaches $l_{max} - l_{nh}$, that is, maximum possible packet length ($1500\,\mathrm{B}$) without the header, bandwidth utilisation is at its optimum with $r_2 \frac{\Phi}{l_{max}}$. Lowering Φ splits the payload into more packets, which generates more overhead and therefore directly consumes extra bandwidth. Because latency in a packetiser is determined by the maximum packet length over rate, lowering Φ also lowers latency.

Impact of Timeout The necessity of $T \geq \frac{\Phi}{r_2}$ immediately becomes obvious. If T was less, the packetiser could never reach Φ before hitting the timeout and so a lower limit for T has been found. We shall now establish a sensible upper boundary for T.

A packet size l processed over a timespan T is an expression of bandwidth. Φ being set, we can now employ T to choose the bandwidth. Because the DM is the only source of high priority traffic, unused bandwidth is equivalent to bandwidth lost to overhead. The proposition is therefore that it is allowable to introduce additional overhead without changing maximum throughput, as long as the combined traffic does not exceed the maximum bandwidth at the system's bottleneck.

The bottleneck is encountered at the WR network, the switches having the lowest rate at r_1. This is in turn down scaled by a factor S_F^{-1} because, as discussed in section 6.7, forward error correction will introduce further redundancy. The resulting bandwidth is denoted as the system's sustainable rate r_s. This can be used to propose the limits for T:

$$\frac{\Phi}{r_2} \leq T \leq \frac{\Phi + l_{nh}}{r_s} \tag{6.45}$$

The reason is that if a maximum sized payload plus its header can be processed at the systems sustainable rate within the timeout, any reduction in payload flow will create more overhead. However, if the sum stays within the sustainable rate, backlog from overhead cannot accumulate. So $T = \frac{\Phi + l_{nh}}{r_s}$ would ensue an optimal bandwidth utilisation. In the absence of any other high priority source, eq. 6.43 and 6.44 can be simplified to:

$$\mathrm{S_{Ps}(a)} = \frac{\Phi + l_{nh}}{\Phi} \tag{6.46}$$

$$\mathrm{S_{Ps}^{-1}(a)} = \frac{\Phi}{\Phi + l_{nh}} \tag{6.47}$$

If any lower latency is required, it can be obtained at the loss of bandwidth to overhead by decreasing T_{tx}.

Absolute Figures The total delay budget for the system was given as $\Delta_t = 500\,\mu s$, so it is interesting whether to test the obtained boundaries chosen for T actually fit within this frame.

$$T_{min} = \frac{\Phi}{r_2} = \frac{1440\,\mathrm{B}}{2\,\mathrm{Gbit\,s^{-1}}} = 5.76\,\mu\mathrm{s} \tag{6.48}$$

$$T_{max} = \frac{\Phi + l_{nh}}{S_F^{-1}(r_1)} = \frac{1496\,\mathrm{B}}{0.25 \cdot 1\,\mathrm{Gbit\,s^{-1}}} = 47.78\,\mu\mathrm{s} \tag{6.49}$$

At 1.2% of the total delay budget, we can assume T_{min} to be a safe choice. T_{max} however, at 9.5%, should be examined again in the final analysis.

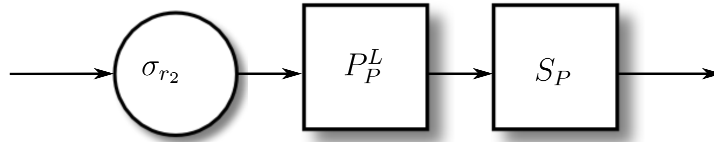


Figure 6.15: Block Diagram of EBM TX Module

Service The minimum (and because of the fixed timeout also maximum) service curve for the TX module is that of a standard packetiser, but the delay is solely determined by T. Note that scaling is applied last and therefore not part of the service curve of this module. The resulting curve is thus very straightforward:

$$\beta_{tx} = \beta_{r_2,T_{tx}} \tag{6.50}$$

6.7 FEC

Context and Necessity The CS ultimately needs a central instance (separate, but synchronised instances are just an equivalent model) which communicates that servers providing the physical calculations to control the accelerator. This again means that there must be a fan-out from the CS's DM to numerous endpoints, placing it as the root node of one or more networks of a tree topology.

The concept of the current CS relies on treating the whole system and timing network as lossless. If it was not, commands would need to be re-sent if they did not arrive at their destination, which adds the need for

feedback from the endpoints. Due to the tree topology, this is already a problem because the bottleneck on the way up to the root node is getting ever tighter and a reason to avoid re-transmission. The second reason is that an upper bound on control loop delay is only possible if re-transmission does not need to be considered.

Application The chances of packet loss has to be reduced to a level at which, for all practical purposes, the system can be treated as lossless. The way to achieve this involves both increasing the system's mean-time-between-failure and employing forward error correction algorithms. The purpose of the latter is to protect both network packets *and* their meta information against bit errors, more details can be found in the work of **prados_boda_fec_2010**, **p**

Impact on the Model In the scope of this work, the Forward Error Correction (FEC) will be treated as a black box system. The observable effect is the creation of k interleaved packets from one incoming packet after a packetisation and encoding delay.

6.7.1 FEC Encoding

The FEC re-packetises to the length set in the EBM TX modules, then starts the encoding process, buffers the resulting encoded packets and finally sends the encoded (scaled) data.

Packetiser Because the packets are of variable size, the first packetiser does introduce a delay equal to $\frac{l_{max}}{r_2}$. After encoding, the second packetiser has to deal with the scaled version of the packets, thus adding a delay of $k \cdot \frac{l_{max}}{r_2}$.

Encoding Time We will assume a constant encoding time in the FEC, introducing a delay of T_e, which already includes the introduced $k-1$ inter-frame gaps between the generated packets. Additionally it is assumed that a time $T_d \geq T_e$ is required to decode the information again in the TR.

Scaling Data will be scaled up to mimic the FECs introduction of redundant data. Similar to the EBM framer in 6.5, this can be described by the application of a scaling curve for the FEC, S_F, which multiplies packet size by k, the number of packets generated.

The resulting sub-system, packetiser, delay and scaler and output packetiser, is shown in figure 6.16.

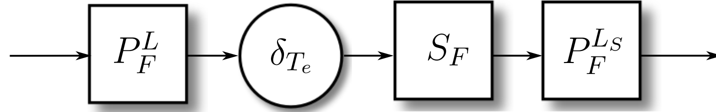


Figure 6.16: Block Diagram of FEC Module

The resulting scaling functions are:

$$
\begin{aligned}
\mathrm{S_F}(\mathrm{a}) &= k \cdot a \\
\mathrm{S_F^{-1}}(\mathrm{a}) &= \frac{1}{k} \cdot a
\end{aligned}
\tag{6.51}
$$

Service The rightmost packetiser in figure 6.16 is expecting the scaled data. We can therefore apply the scaling function to the maximum packet size and obtain the following service curve for the FEC:

$$
\beta_{\mathrm{F1}} = \beta_{r_2, \frac{l_{max}}{r_2}} \otimes \delta_{T_e} \otimes \beta_{r_2, \frac{S_F(l_{max})}{r_2}} \tag{6.52}
$$

6.7.2 TR

FEC Decoding Interest lies in determining an *end-to-end* delay bound for the timing system. In the present case, the decoding happens in the timing endpoint, which is why it is necessary to also model this part of the TR in some detail.

Gathering Packets and Decoding Time For forward error correction, only a part of the packets belonging to one transmission need to arrive. It is necessary to assume the worst case though, which means waiting for the last packet to arrive. The first step is therefore re-packetising all arriving packets from one transmission into one big packet. Afterwards, the packets are decoded, which makes the whole decoder use the same equation as the encoder. The service curve is similar to eq. 6.52.

Symmetric Scaling As already presented in section 6.2.4, a symmetric scaling variant is applied, which requires the corresponding decoder to apply the inverse of the original scaling function.

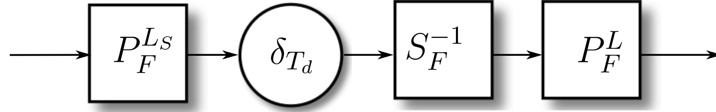


Figure 6.17: Block Diagram of the decoder Module

$$\beta_{F2} = \beta_{r_2, \frac{S_F(l_{max})}{r_2}} \otimes \delta_{T_d} \otimes \beta_{r_2, \frac{l_{max}}{r_2}} \tag{6.53}$$

6.8 Etherbone Slave and Event-Condition-Action Unit

Demultiplexing Removal of the packet header happens instantaneous at the inverse scaler. So the EBS Receiver (RX) block diagram looks like a mirrored version of the EBM TX (see figure 6.15 and 6.18). The EBS de-framer does not work the same way as the EBM framer, it removes the EB record header and adds a delay, but does not re-packetise. This can therefore be described as an inverse scaler followed by a rate-latency system (see lower right of figure 6.18, "EBS Deframer").

This finally leaves the Event Condition Action unit (ECA) unit (see **??**). Being one of the most complex logic cores in the system, it can nevertheless be described as a simple black box model. ECA is responsible to schedule actions originating from arriving messages to be executed at the timestamp they carry. To sort arrivals, ECA adds a bounded delay of 4 µs.

This would be followed by an EDF scheduler, which is not modelled. The reason is that all messages are dispatched $\Delta_t = 500$ µs before they are due. For all messages that arrive on time, the EDF would hold them back until their execution time. This would hide the leftover delay budget from the analysis.

6.9 White Rabbit Network Model

6.9.1 Interference at Network Integrated Controller (NIC)

Origin At the network interface, the DM's flow is multiplexed with flows from the WR timing core. WR uses Precision Time Protocol (PTP) packets to synchronise the Coordinated Universal Time (UTC) time between TRs/switches. In addition, there are other services spuriously sending

packets, like the Address Resolution Protocol (ARP), Dynamic Host Configuration Protocol (DHCP) or Simple Network Management Protocol (SNMP).

Approach for DM HP Service DM traffic has the highest priority of all services. However, pre-emption is not allowed, so the minimum service to the DM has to consider waiting for transmission of the longest possible low priority packet. Packet lengths of lower priority services can be described as $l_{\text{<name>}} \sup_{n \in \mathbb{N}}\{l_{\text{<name>}}(n)\}$. The longest possible low priority packet is thus $l^{lo}_{max} = \max\{l_{ptp}, l_{arp}, l_{dhcp}, l_{snmp}\}$, resulting in the DM a minimum service of

$$\beta_{\mathrm{N}} = \beta_{r_1} \ominus l^{lo}_{max} \tag{6.54}$$

Improving WR PTP performance WR PTP periodically has to send packets to synchronise UTC time to counter long term drift against the time reference (Global Positioning System (GPS) receiver). It is questionable if a system with only two priorities is a good design choice because a continued starvation of WR PTP service would lead long periods of uncompensated clock drift in all downstream timing switches and receivers. It would therefore be sensible to introduce another priority level between DM and background for WR PTP, and allocating a minimum service rate for clock synchronisation.

PTP needs periodic adjustment, so it is assumed that all WR PTP flows are periodic. The corresponding staircase functions are sufficient to model the arrival curves **thiran_network_2001**. The presented approach models the mutual influence on service by application of one shaper per flow. This limits the influence of the interfering flow to its maximum allocated rate. Assuming a fraction of the total rate k is assigned, with $k \in \mathbb{N}^*$, to WR traffic. It would then be forced to obey $\sigma_{ptp} = \lambda_{\frac{r_1}{k}}$, making sure α_{ptp} (as an interfering flow) does not exceed this rate in the presence of DM traffic. It follows that the shaper for the DM traffic must guarantee the agreed minimum rate to WR, which leads to $\sigma_{dm} = \lambda_{r_1 \cdot \frac{k-1}{k}}$. The shaping curves are the guaranteed service for each flow, and $\lambda_{r_1} \geq \sigma_{ptp} + \sigma_{dm}$. The maximum length packet to be considered in the delay equation differs for WR and DM flows, as DM is of highest priority, WR of second highest and all others of lowest priority. The leftover service for each of the higher priority flows

can be calculated as:

$$\alpha_{\text{ptp}} = l_{ptp} \cdot \upsilon_{T_{ptp},0} = l_{ptp} \cdot \left\lceil \frac{t+0}{T_{ptp}} \right\rceil = \gamma_{\frac{l_{ptp}}{T_{ptp}}, l_{ptp}} \tag{6.55}$$

$$\beta_{\text{nic}} = \lambda_{r_1} \ominus (\alpha_{ptp} \oslash \sigma_{ptp}) \otimes \delta_{\frac{l_{max}}{r_1}} \tag{6.56}$$

6.9.2 WR Switches

The DM is connected via a tree topology to 2000+ TRs. WR switches feature 18 ports each, which means a fanout of 1-17. This indicates a minimum of $k = \lceil \log_{17} 2000 \rceil = 3$ layers of switches. The topology is adjusted for geographical reasons though, so the actual size is likely to be 5 layers. WR switches are treated as black boxes. The delay they introduce has been removed from traffic measurements and is represented in a simplified model.

Switch Properties All switches treat traffic from the DM as high priority. The switches feature cut-through for low latency. This means HP packets are passed on as soon as possible, sending the first bits before their last bits have arrived. The switches are non-preemptive, meaning they must buffer (introduce a delay) if a lower priority packet is currently being transfered.

The switches are therefore modelled as a small constant delay T_s representing the time it takes to inspect the packet header and apply the switching matrix. Following this is the multiplexer, a constant rate node operating at r_1. Because the switch can be busy with a low priority packet, there is another delay of $\frac{l^{lo}_{max}}{r_1}$ in its minimum service. Because LP traffic is partly point-to-point and originates at all switches, it cannot be assumed multiplexing is applied only at the first switch. The service of a WR switch is therefore defined as:

$$\beta_{\text{sw}} = \delta_{T_s} \otimes (\beta_{r_1} \ominus l^{lo}_{max}) \tag{6.57}$$

6.10 End-to-End Delay Analysis

All sub-components of the CS have been modelled and an end-to-end delay analysis of the complete system can now be conducted. We will perform the necessary preparation for a PMOO analysis, as it (in most cases) leads to the tightest delay bounds. In the scope of this thesis, PMOO

also has the benefit that the required reduction of the system into a single equivalent node allows a clearer visualisation.

For ease of representation, the system is split into four parts. The first is the sink tree posed by CPUs, aggregating flows from their threads, and the PQ, aggregating flows from CPUs. The second is the single-path section of the DM without the NIC, the third are the NICs of DM and TR as well as all WR switches. The fourth and last is the TR.

Packetisers in the Big Picture To reduce the size of the figures, all packetiser blocks in the following diagrams show not just L-packetisers, but already PGS, a combination of a bit-by-bit system and an L-packetiser (see figure 6.13, 6.15). Furthermore, a substitution of L-packetiser functions was applied. Packetiser P_{M_1} has a maximum packet size of $l_p = 32\,\mathrm{B}$. Afterwards, EB record headers are added, the packets are scaled. P_{P_1} collects EB records, which is a timing message scaled with S_R. It has maximum packet (payload) size of $l_{np} = k \cdot S_R(l_p) = S_R(kl_p)$. FEC input collects payloads plus network header, which means scaling them by S_P so maximum packet size is $l_n = S_P(S_R(kl_p))$. The encoder outputs encoded packets, means scaled by S_F, which equals a maximum packet size of $l_f = S_F(S_P(S_R(kl_p)))$. All three L-packetisers can therefore be expressed by the same L-packetiser and a scaling function. In the block diagram, all L-packetiser scaling is noted *above* the PGS' name and any scaling applying to the bit-by-bit system *below* the PGS' name.

Network Tunnel Because flows of timing messages are aggregated into network packets and separated at the endpoints EBS, they intermittently become a single flow in the model. This is called a trunk or tunnelled connection and is marked in grey in the following overview figure 6.18. More details can be found under subsection 6.10.5.

6.10.1 EDF Sink Tree

Depending on whether the flow of interest for PMOO analysis is defined as all timing message input flows or a single one, different service and arrival curves for CPU and PQ have to be used. We shall designate a flow of interest with ingress at the CPU level as R_{xy}, being the x-th flow at CPU y.

Sum of all Timing Flows If the intention is finding the delay bound for any and all of the timing flows, the arrival curves of all flows must be

aggregated and the service of the PQ's constant rate note is concatenated with the single path section of the system, denoted as β_{sp}. The incoming flow at the PQ node is then defined by

$$\alpha_{\mathrm{pq}} = \sum_{j}^{N-1} \sum_{i}^{M-1} \left(\alpha_{ij} \oslash \left(\beta^{CPU} \otimes \beta_{q} \otimes \delta_{a} \right) \right) \tag{6.58}$$

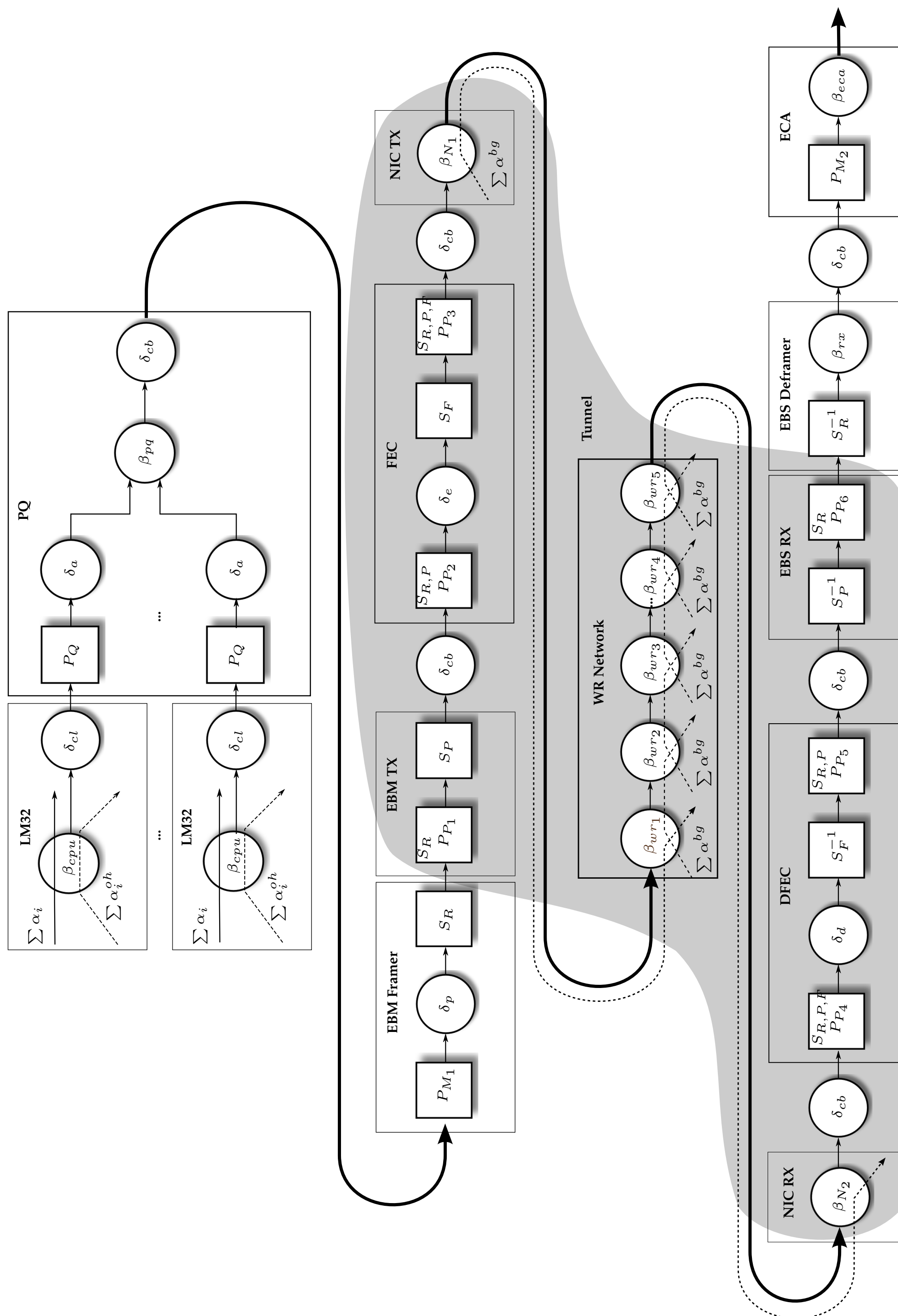


Figure 6.18: Block Diagram of NC CS Model:
Data Master (1st and 2nd row), White Rabbit Network (3rd row), TR (4th row). Tunnel coverage is shown in grey

Single Flow of Interest The delay bound for a single timing message flow can be obtained by calculating the leftover service at both the CPU at which the FOI originates and the PQ. We shall start by modifying eq. 6.58 to include only the flows originating at *other* CPUs by replacing the limits of the first sum by $j \in [0, N) - \{y\}$, with y being the index of the origin CPU of the FOI.

$$\alpha^*_{-\mathrm{y}} = \sum_{j \neq y}^{N-1} \sum_{i}^{M-1} \left(\alpha_{ij} \oslash \left(\beta^{CPU} \otimes \beta_q \otimes \delta_a \right) \right) \tag{6.59}$$

We then need to add all flows originating at CPU y, except for the FOI x, and calculate the matching output arrival curve by feeding the aggregate flow through the CPUs leftover service, after serving the FOI:

$$\beta^{\mathrm{l.o.}}_{-\mathrm{xy}} = \left(\beta^{CPU} \ominus \left(\sum_{i}^{M-1} \alpha_i^{oh} + \alpha_{xy} \right) \right) \otimes \beta_q \otimes \delta_a \tag{6.60}$$

$$\alpha^*_{-\mathrm{xy}} = \sum_{i \neq x} \alpha_{iy} \oslash \beta^{l.o.}_{-xy} \tag{6.61}$$

This leaves concatenating the leftover service the FOI experienced at both CPU and PQ level. The leftover service for the FOI at CPU level is given by eq. 6.27 on p. 79. Combined with eq. 6.59 and 6.61, the leftover service in the sink tree is

$$\beta^{\mathrm{l.o.}}_{\mathrm{st}} = \lambda_{r3} \ominus \left(\alpha^*_{-y} + \alpha^*_{-xy} \right) \tag{6.62}$$

6.10.2 Equivalent Circuit for WR Network

The WR network consists of several layers of WR switches for which an equivalent node needs to be created. The NIC nodes of DM and TR are also to be combined into the equivalent system, because they also carry the interfering WR background flows (PTP, DHCP, ARP, SNMP). The middle row in figure 6.18 shows the structure of the WR system. An equivalent node is therefore the leftover service in the concatenation of all involved nodes. Since DM traffic is high priority and non-preemptive, all nodes must allow for a maximum length low priority packet to complete. For the WR switches 6.57 and the NICs 6.54, this is already included.

It is noteworthy that the calculation of leftover service in the present case does pay for multiplexing several times. This is done on purpose, because WR background traffic is generated at all switch levels and the NICs.

Therefore, multiplexing *does* happen at each node again. The equivalent service for all WR network nodes can be written as

$$\beta_{\text{wr}} = \beta_{N1} \otimes \bigotimes_{1 \leq i \leq k} \beta_{sw} \otimes \beta_{N2} \tag{6.63}$$

6.10.3 Data Master to Timing Endpoint

Concatenation and Scalers The presence of scalers in the system (all S_x blocks in figure 6.18) prevents direct analysis. Nodes separated by a scalers cannot be concatenated by standard NC, so the scalers need to be removed. This can be achieved by one of three ways:

- Move all scalers to ingress
- Move all scalers to egress
- Move symmetric scalers to their inverse

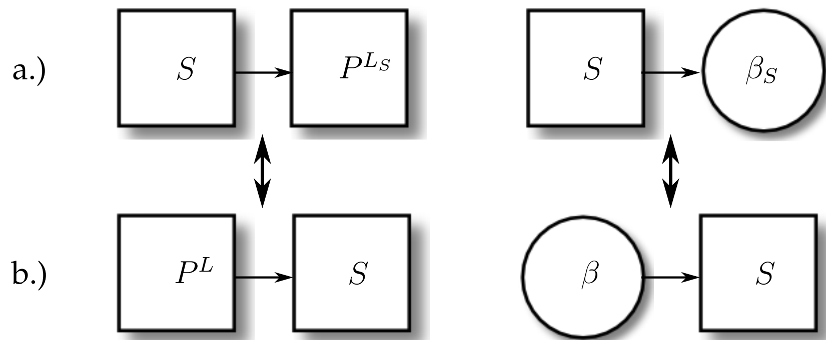


Figure 6.19: Equivalent Circuits for Scalers

Moving Scalers The present case has symmetric scalers, so it is possible to move them towards their respective inverse functions (S_x adjacent to S_x^{-1}) so they cancel each other out. Moving a scaler is achieved by replacing it with its equivalent circuit from figure 6.19. The replacement follows the rules for scaled service on p. 67 in section 6.2.4, the same principle holds for L-packetisers and their scaling functions.

The tightness of the achieved bounds depends on the chosen equivalent circuits. The best choice differs for backlog, output and delay bounds. **fidler_way_2006** states that for delay analysis, a system in the a.) row of figure 6.19 should stay unchanged, and a system from the b.) row can be

changed to a.). This means the three scalers S_R, S_P and S_F in the DM are to be moved downstream until they reach S_R^{-1}, S_P^{-1} and S_F^{-1} and cancel each other out.

Step-by-Step Removal While the presented method removes the scaler blocks, it is clear that their influence on other components remains. The removal process is described using the notation for packetisers and their bit-by-bit systems from p. 93.

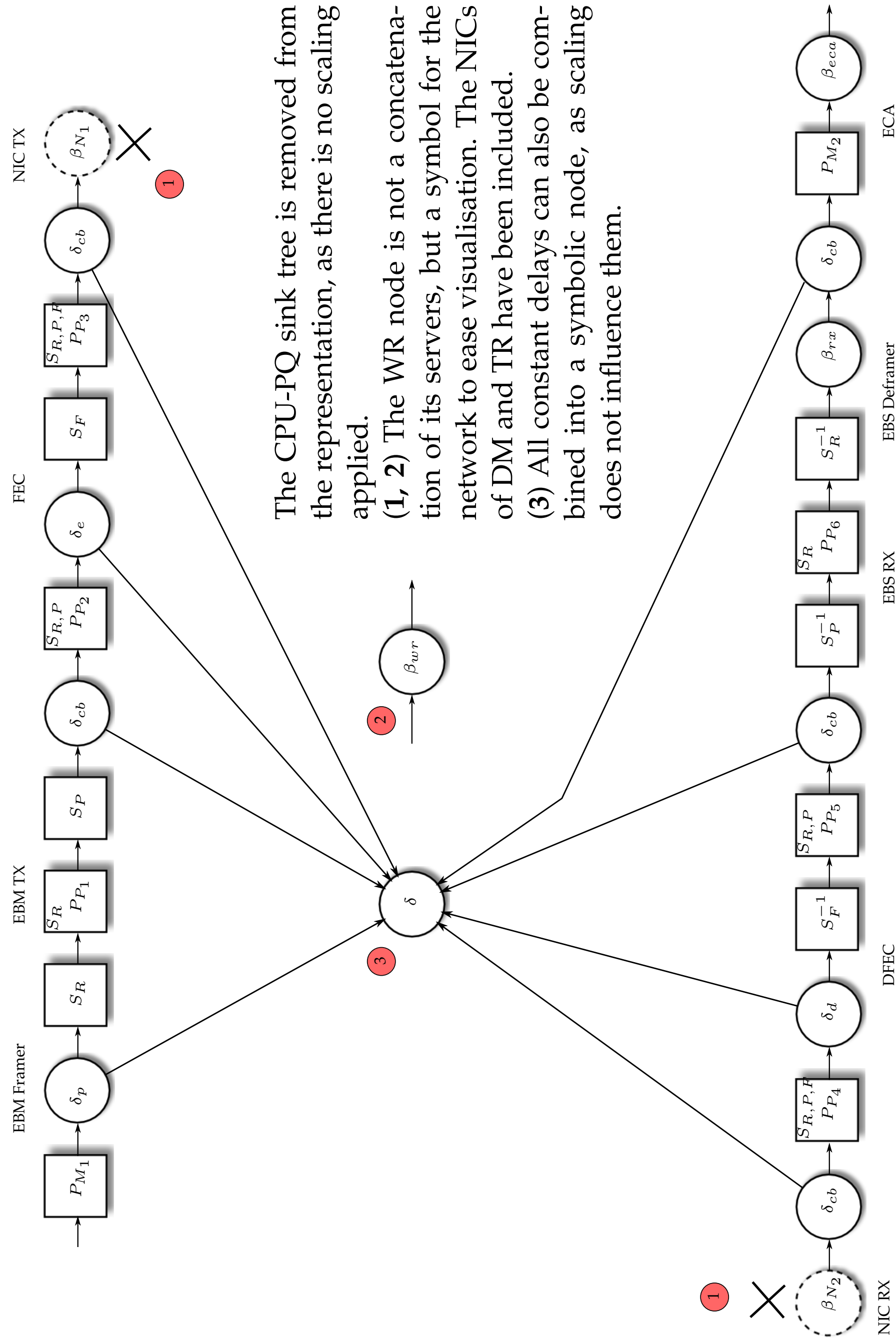


Figure 6.20: Introduction of Symbols for static Delays and WR Network

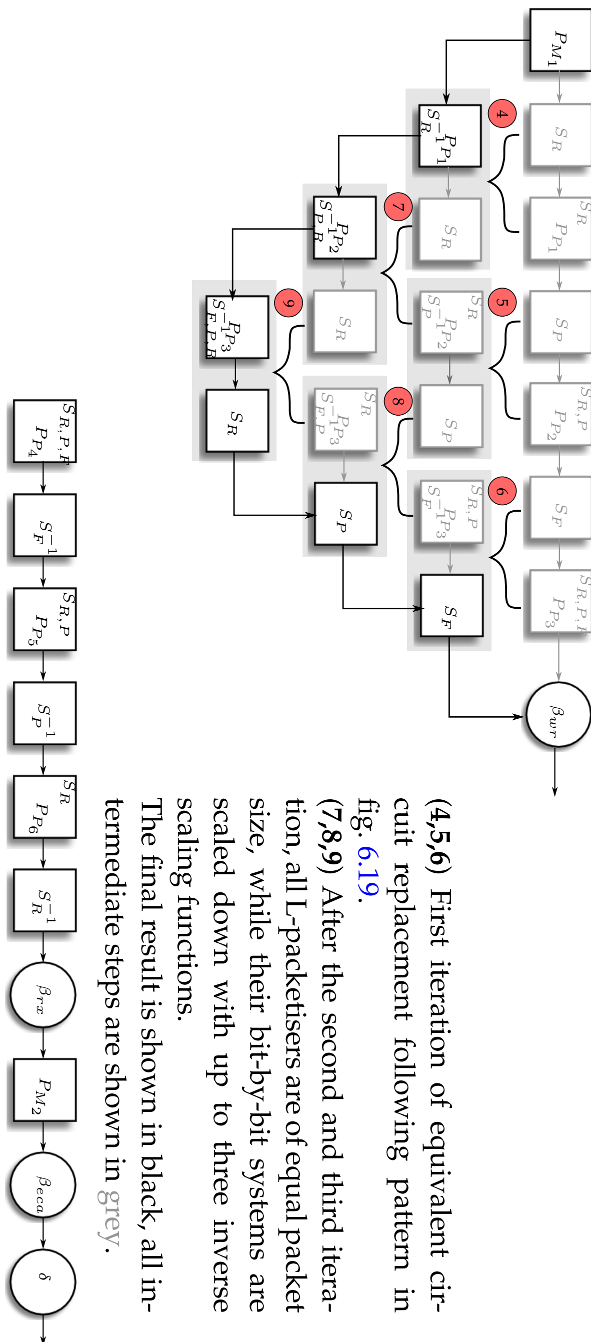


Figure 6.21: Replacement of DM Scalers

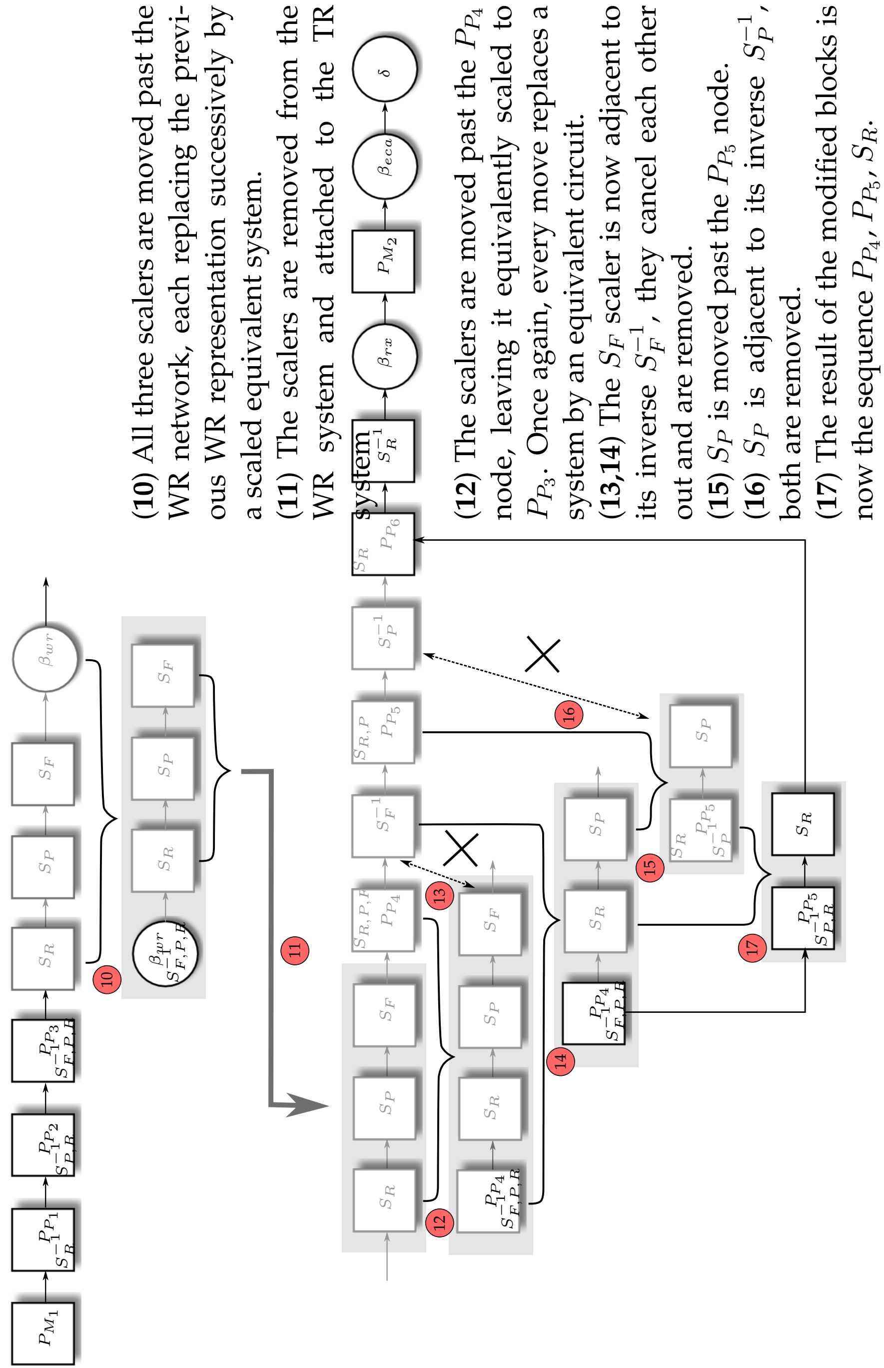


Figure 6.22: Scaling WR Network (NW) and Replacement of TR Scalers

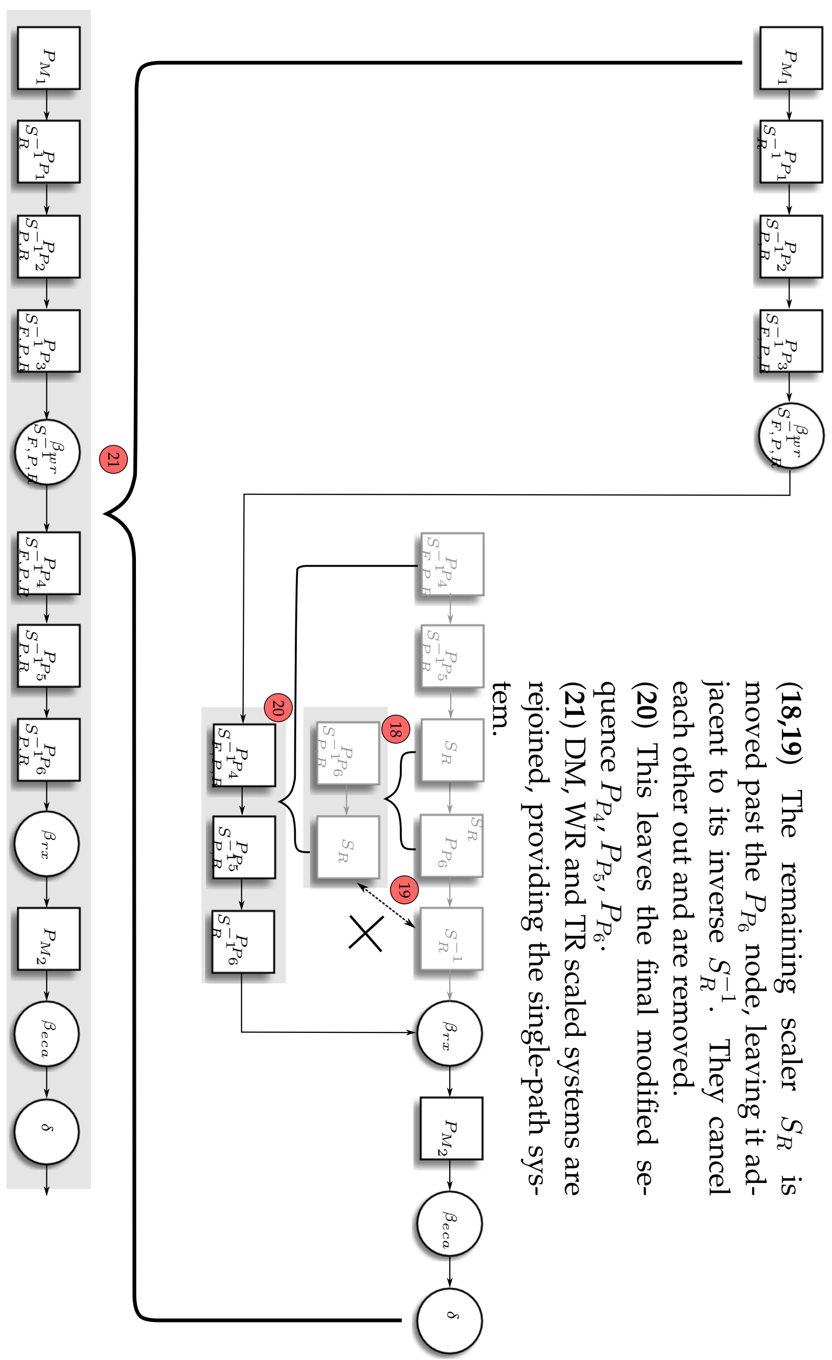


Figure 6.23: Final TR Scaler Replacement and Join with DM Blocks

6.10.4 Equivalent Service of Packetised Greedy Shapers

Moving a scaler downstream past a packetised greedy shaper will influence both the L-packetiser and the bit-by-bit system. Thus, it will transform a system of the form σ, P^{L_S} into an equivalent circuit of $\underline{S^{-1}}(\sigma), P^L$.

Iterative Scaling Because the minimal scaling curve (max-plus deconvolution) of scaling functions are defined to represent less or equal service, it follows that iterative use drives the curves towards pessimistic service representations, so

$$(S_X \overline{\oslash} S_X)((S_Y \overline{\oslash} S_Y)(a)) \;\leq\; S_X(S_Y(A)) \overline{\oslash} S_X(S_Y(a))$$

Therefore the process followed is to firstly apply all scaling functions, i.e. $S_X(S_Y(S_Z(a)) \ldots$, then apply the deconvolution operator. Because the scaling functions are bijective, the order is of no consequence. To provide a more readable representation, the following notation is used:

$$\begin{aligned} \mathrm{S}_1(\mathrm{a}) &= S_R(a) \\ \mathrm{S}_2(\mathrm{a}) &= S_R(S_P(a)) \\ \mathrm{S}_3(\mathrm{a}) &= S_R(S_P(S_F(a))) \end{aligned}$$

Using these shortforms, the equivalent service of all packetisers in block diagram 6.23 is given by the following equations:

$$\mathrm{P}_{\mathrm{M}_1} = \mathrm{P}_{\mathrm{M}_2} \qquad \to \beta_{r_2, \frac{l_p}{r_2}} \tag{6.64}$$

$$\mathrm{P}_{\mathrm{P}_1} = \mathrm{P}_{\mathrm{P}_6} \qquad \to \underline{S_1^{-1}}(\beta_P) \qquad \leq \beta_{S_1^{-1}(r_2), \frac{kl_p}{S_1^{-1}(r_2)}} \tag{6.65}$$

$$\mathrm{P}_{\mathrm{P}_2} = \mathrm{P}_{\mathrm{P}_5} \qquad \to \underline{S_2^{-1}}(\beta_P) \qquad \leq \beta_{S_2^{-1}(r_2), \frac{kl_p}{S_2^{-1}(r_2)}} \tag{6.66}$$

$$\mathrm{P}_{\mathrm{P}_3} = \mathrm{P}_{\mathrm{P}_4} \qquad \to \underline{S_3^{-1}}(\beta_P) \qquad \leq \beta_{S_3^{-1}(r_2), \frac{kl_p}{S_3^{-1}(r_2)}} \tag{6.67}$$

For the sake of completeness, a shorthand scaled version of the WR service reads

$$\beta_{\mathrm{wr}_\mathrm{s}} = \underline{S_3^{-1}}(\beta_{wr}) \tag{6.68}$$

6.10.5 Aggregate Scheduling

The system has been modelled in terms of its service, but there is a hitherto unconsidered constraint on flows traversing the system. When flows

of timing messages are multiplexed on the DM's WB bus, this follows a standard NC model. Between the DM and the endpoint however, this becomes an Ethernet based network connection.

Aggregation From the EB TX module to the EB RX module, messages are bundled into network packets (grey area in figure 6.18). This wrapping is called aggregate scheduling, or in more general networking terms, a tunnelled connection. There is a significant difference in the multiplexing behaviour, because the multiple timing message flows become one single flow of network packages while they are in the tunnel. Message flows are no longer running as cross traffic to each other and thus cannot delay each other. The resulting single flow only has the WR services as its cross traffic and because DM traffic is treated as HP without pre-emption within the WR network, only the processing times for maximum length LP traffic accumulate as latency. As a result, the latency for traversing the tunnel, and therefore the end-to-end delay, strongly decreases (compare chapter **??**, figure **??** and **??**).

For TFA, this is of no consequence, as it operates on the assumption that all incoming flows are aggregated (added) before analysis. For SFA and PMOO however, the ingress, the tunnel and the egress must be analysed as separate cases. The transition between the ingress and the tunnel is trivial, because the arrival curve entering the tunnel is the aggregate, i.e. sum of all the ingress's output arrival curves:

$$\alpha_{\mathrm{t}} = \sum \alpha^{*}_{in_i} \tag{6.69}$$

Regaining Individual Flows Once the tunnel ends at the EB RX module though, a problem presents itself. The output aggregate flow constrained by α_t must now be split up again into the original number of flows, which is not as trivial as it might seem. For the demonstration cases in chapter **??**, the problem can be circumvented because all incoming flows were chosen to be equal. Thus, the output arrival curve of the last node in the tunnel can be divided by the number of original flows. Finding a generic solution for the corresponding residual service curves is not trivial and still work in progress in the beginning of 2017 (see chapter **??**). However, there exists usable workaround for the present case. **bondorf_improving_2016** proved in **bondorf_improving_2016** that the maximum backlog encountered in a TFA at a given node is also the maximum backlog any other form of analysis can encounter, thus capping the backlog. Furthermore, the sustainable rates of the individual flows cannot have increased inside

the tunnel. If one consequently assigns the rates of the ingress's output arrival curves $\alpha^*_{in_i}$ to α_{out_i} and sets their initial burst to the TFA bounded burst value of the aggregate output arrival curve α^*_t, all α_{out_i} are defined by valid arrival curves. From the rate and burst limits, it follows that the resulting arrival curves must be greater or equal to the tight arrival bound, which means they are still valid constraints to their flows.

This allows to obtain an end-to-end delay by adding the individual delay for ingress, tunnel and egress. The calculated difference in latency between assigning the best case (zero burstiness) and the worst case (aggregate burstiness) to any or all output arrival curves causes latency differences in the single digit microsecond range in the DM simulation. The approach was therefore considered an acceptable intermediate solution for the present case.

6.10.6 Summary

In this chapter, it has been shown that NC is applicable to the case study and how its peculiarities can be handled. Additionally, it has been shown that machine schedules, which control accelerator components in the FAIR case study, can be modelled as network flows. Changing flows can be expressed by using suprema of alternative arrival curves or recurring analyses with non-empty buffers.

The model has been further enhanced to show how the trinity of Program, Cycle based Bus and packet based Network of the SoC System can be modelled in NC. All sub-modules have been discussed in detail and service representations have been deduced. The findings were then combined to produce a single equivalent service, which can be used to calculate the maximum delay for a particular flow of interest or the sum of all flows.

In the evaluation in chapter **??**, the results obtained by simulating the model in the Disco DNC v2 simulator **bondorf_discodnc_2014** will be represented and the results, as far as feasibly possible, compared with tests of the prototype system.

Chapter 7

Memory Maps

Lorem ipsum dolor sit amet... blabla

Reserved / Unused
DM RW only
Host RW only
DM RW, Host RO
DM RO, Host RW
DM RW, Host RW

Figure 7.1: Color Legend for Memory Layout

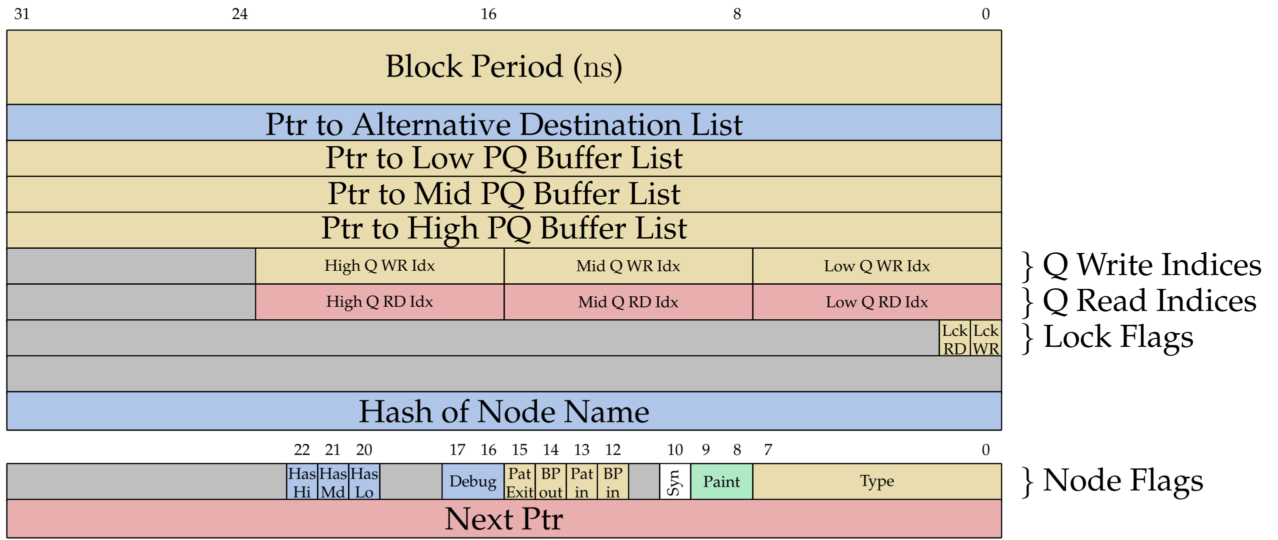


Figure 7.2: Block Memory Layout

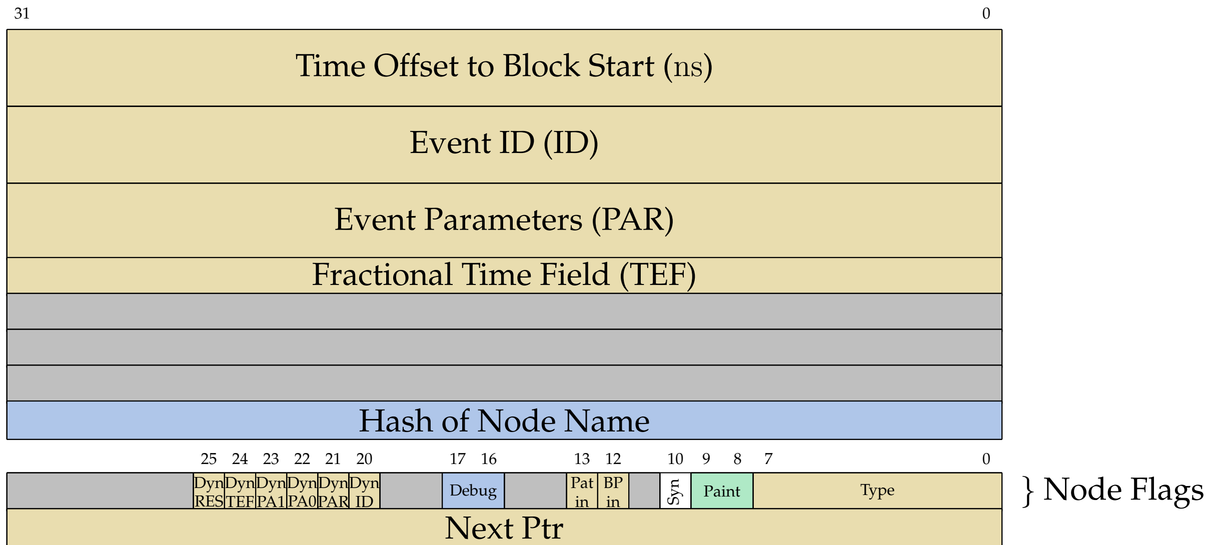


Figure 7.3: Timing Message Memory Layout

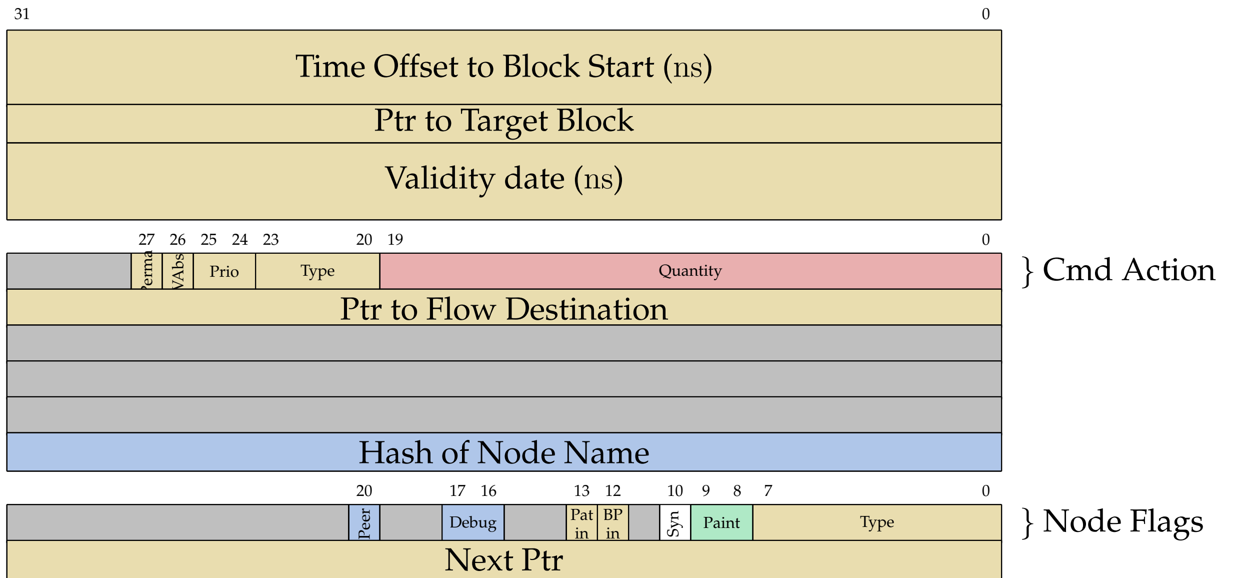


Figure 7.4: Flow Command Memory Layout

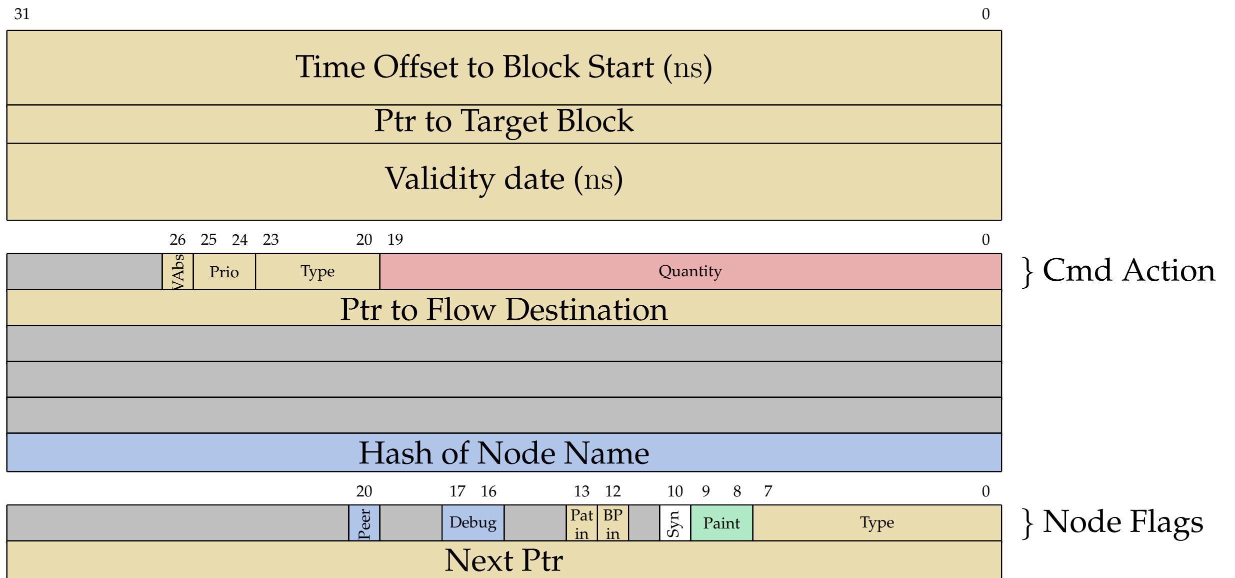


Figure 7.5: No-Op Command Memory Layout

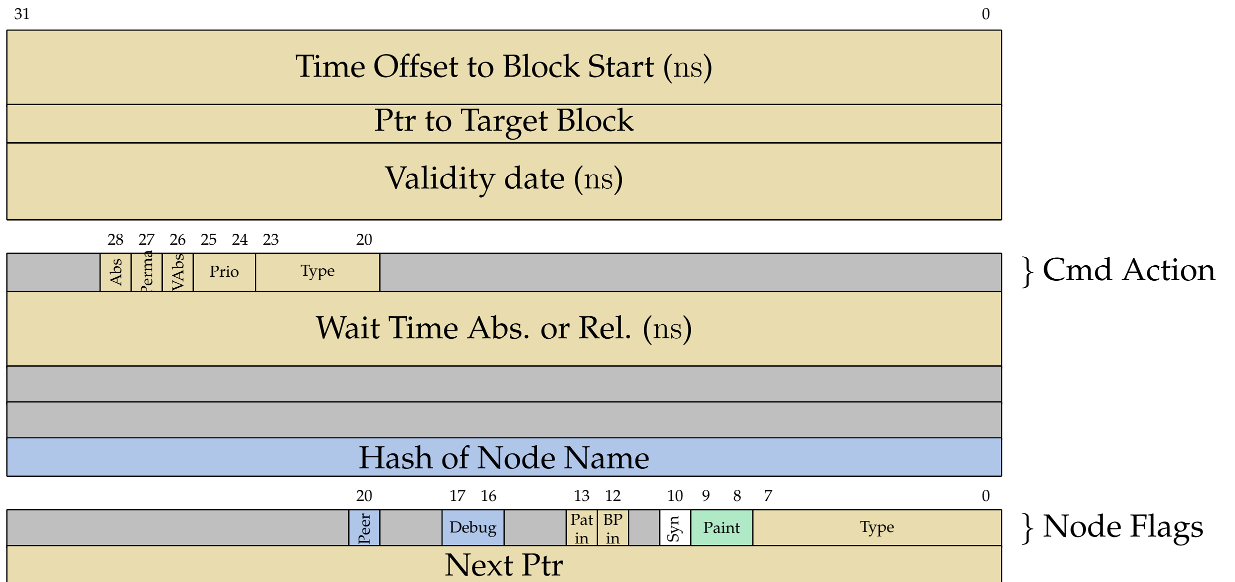


Figure 7.6: Wait Command Memory Layout

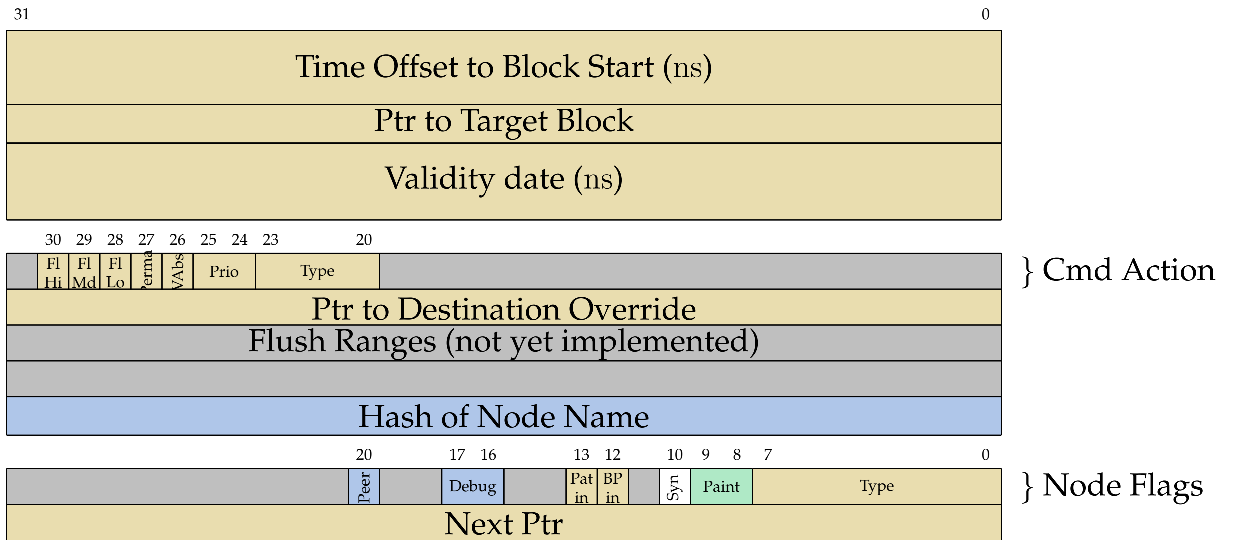


Figure 7.7: Flush Command Memory Layout

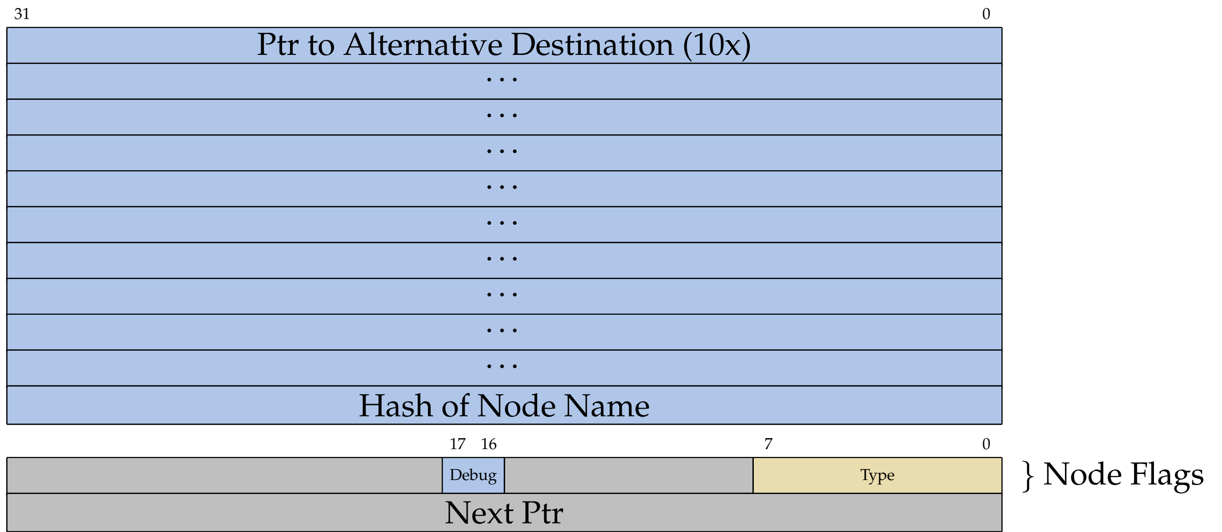


Figure 7.8: Alternative Destination List Memory Layout

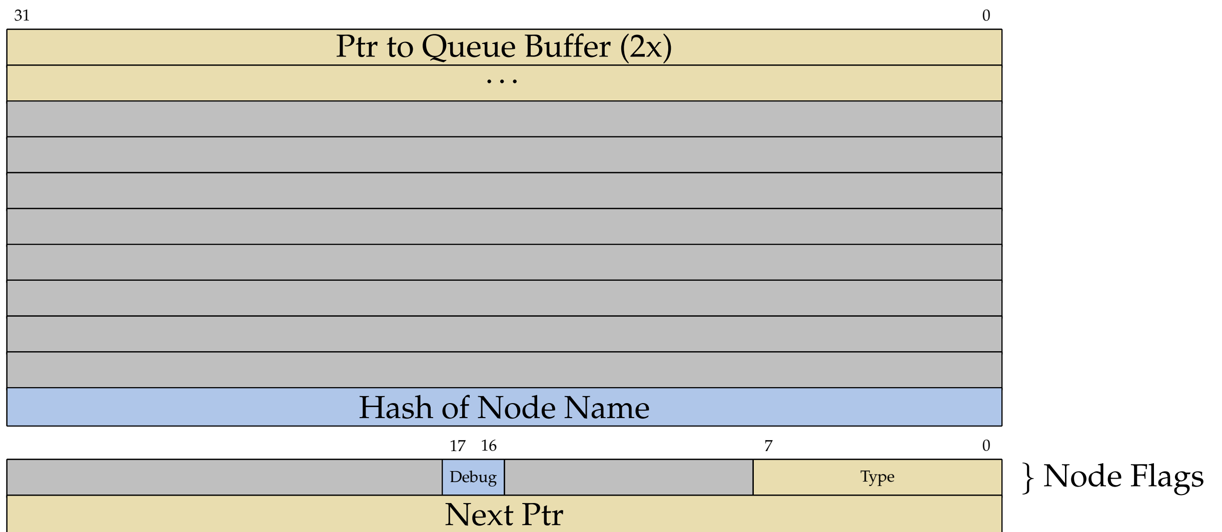


Figure 7.9: Queue Buffer List Memory Layout

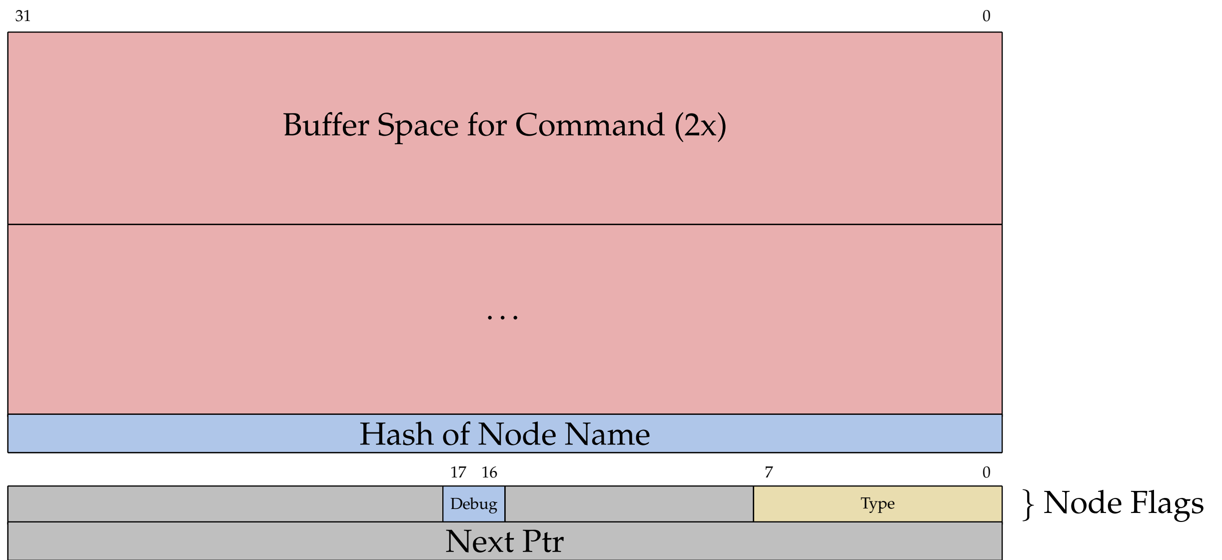


Figure 7.10: Queue Buffer Memory Layout

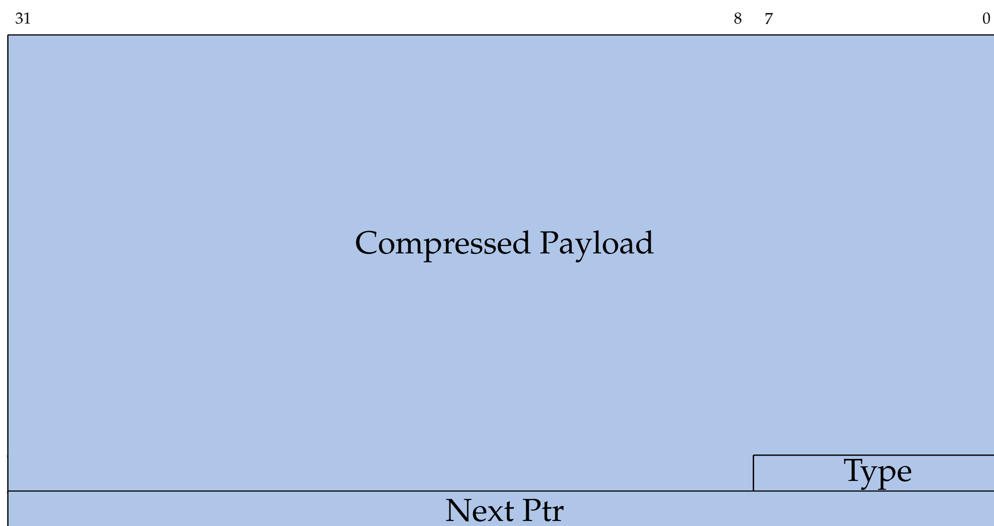


Figure 7.11: Management Chunk Memory Layout

Address range	Register group	Description
0x500 – 0x51f	SHCTL_HEAP	EDF Scheduler Heap and Metadata
0x520 – 0x523	SHCTL_STATUS	Global Status Register
0x524 – 0x52f	SHCTL_META	Management LList Metadata
0x530 – 0x5a7	SHCTL_DIAG	Scheduler Diagnostics and Change Log, see fig. 7.13
0x5a8 – 0x5ab	SHCTL_CMD	Global Command Register
0x5ac – 0x5b3	SHCTL_TGATHER	Time the HW PQ gathers messages for a packet
0x5b4 – 0x5c3	SHCTL_THR_CTL	Global Thread Control and Status, see fig. 7.14
0x5c4 – 0x683 / ⋮ / 0x683	SHCTL_THR_STA	n · thread staging data, see fig. 7.15
0x684 – ⋮ – 0x783	SHCTL_THR_DAT	n · thread runtime data, see fig. 7.16
0x784 – ⋮ – 0x7d7	SHCTL_INBOXES	n · thread inboxes

Figure 7.12: Control Register Groups (space not to scale)

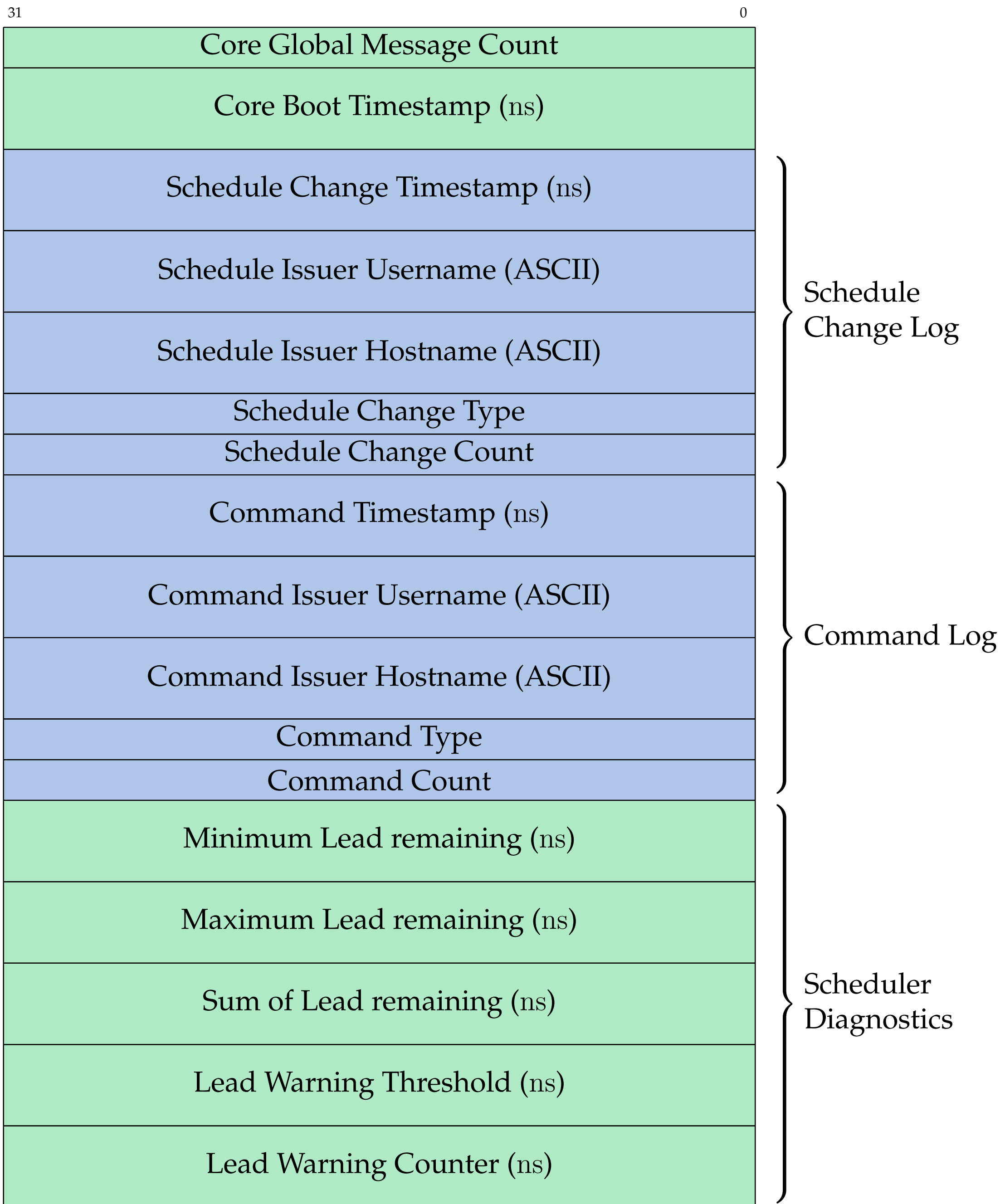


Figure 7.13: Diagnostic and Log Subfields

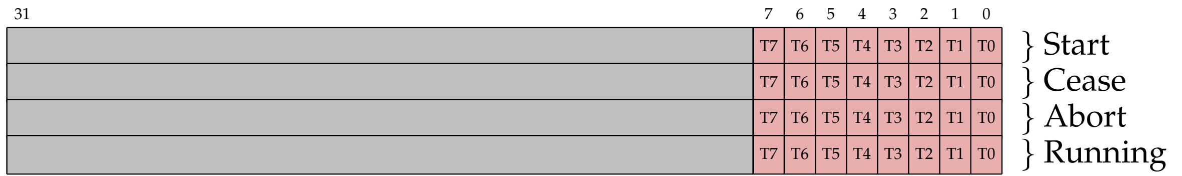


Figure 7.14: Global Thread Control and Status Bits

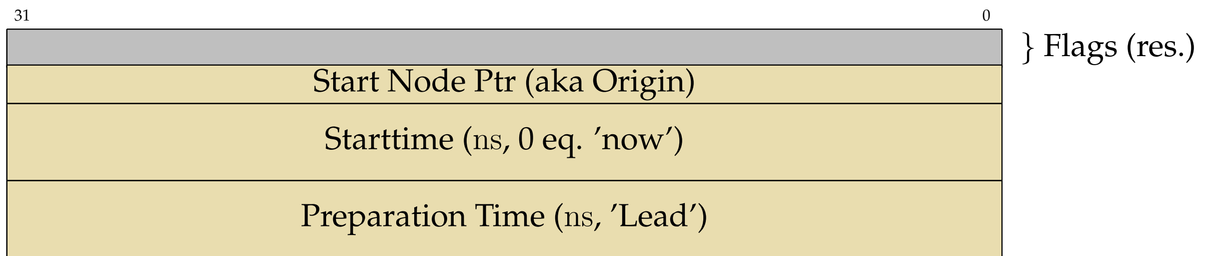


Figure 7.15: Individual Thread Staging Data

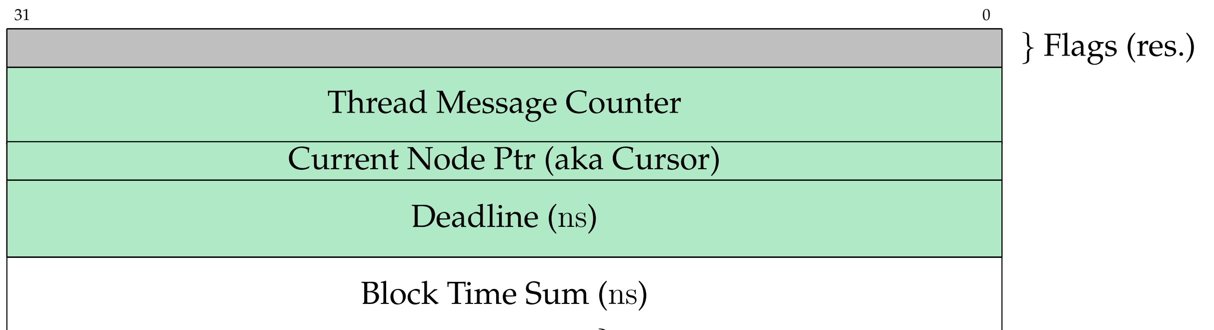
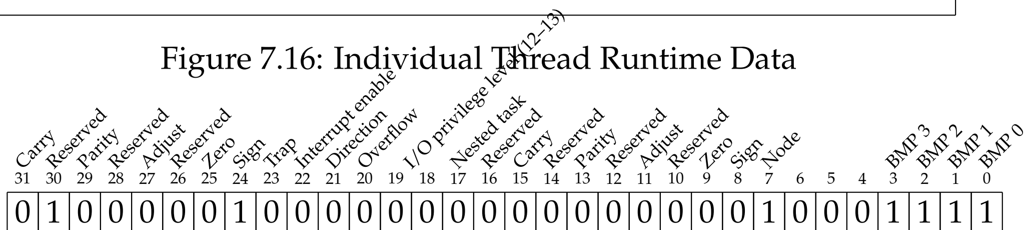


Figure 7.16: Individual Thread Runtime Data



Chapter 8

Quick Reference

Node Attributes

beamproc The name of the beamprocess this node belongs to. Currently not supported.

bpentry If true, this node is an entry point to the beamprocess it belongs to. Currently not supported.

bpexit If true, this node is an exit point from the beamprocess it belongs to. Currently not supported.

cpu Index of the CPU core this Node will reside in. To be replaced by auto-balancer algorithm.

flags Aggregation of certain node flags, internal use only.

id Addresses the 64b ID field of a timing event. Sub-Id fields can be used instead, see id.

node_id Tag of the node's name. Internal use only.

par Transparent parameter field, part of a timing message. Is passed through to a listening ECA channel in timing receiver.

patentry If true, this node is an entry point to the pattern it belongs to.

patexit If true, this node is an exit point from the pattern it belongs to.

pattern The name of thepattern this node belongs to.

permanent If true, the corresponding flow command will permanently change the default destination of the target block, else the change lasts only during the execution of the command.

prio Chooses the command queue level this command will be written to at the target block. (0 (lo), 1(mid), 2(hi)).

qhi Block attribute, generates a command queue of priority Mid if true.

qil Block attribute, generates a command queue of priority High if true.

qlo Block attribute, generates a command queue of priority Low if true.

qty Repetition quantity of a command(-node). The generated command will be executed until qty reaches 0.

tef Fractional (sub-nanosecond) time extension field, part of a timing message. Currently not interpreted by ECA).

thread Index of the thread core this Node will be handled by. To be replaced by auto-balancer algorithm.

toffs The time offset ns relative to the block start at which this node is executed. Negative offsets can be used for debugging to force late events.

tperiod The duration of a block. Amount of time in ns it adds to the current time sum when processed.

tvalid Time in ns at/after which this command will be valid (executable), can be absolute or relative depending on vabs flag.

twait Timespan in ns a wait command is goid to wait. Can absolute or relative to current time, depending on flag.

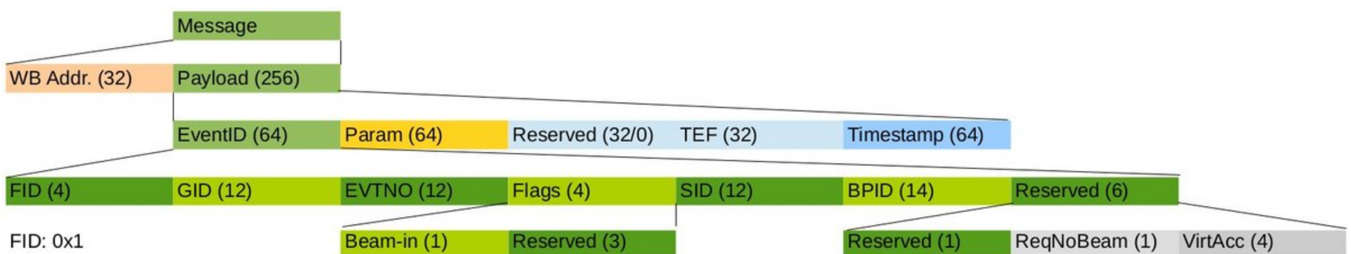
type Determines the type of this node (e.g. block, tmsg, etc.).

- **block** Fixed length block. Terminates a sequence of nodes and defines the sequence's length in time. See tperiod.
- **blockalign** Alignment block. Extends its own length so the time sum will become a multiple of the time grid (currently 10 µs). See tperiod, block.
- **flow** Causes one or more repetitions of a flow command to be written to the target block's queue, changing it's default successor (e.g. branch, loop, etc).
- **flush** Causes a flush command to be written to the target block's queue, emptying any selected queues. Can optionally also override the default successor.
- **nop** Causes one or more No Operation command(s) to be written to the target block's queue. No direct effect, but can together with flow create a sequence of default and alternative successors.
- **tmsg** Timing Message. Causes a message to be broadcasted to all timing receivers.
- **wait** Causes a wait command to be written to the target block's queue, temporarily stretching it's length in time.

vabs Chooses whether tvalid is interpreted as absolute value or an offset to current time sum. When using this as a loop initialiser head, *always* use vabs=true and tvalid=0!.

Timing ID sub fields

beamin	Marks this node as an exit point to beamprocess X. A node can belong to only one beamprocess, part of the ID field of a timing event..
bpid	ID of the beamprocess this timing event belongs to.
evtno	The event number this timing event belongs to.
fid	Format this timing event ID follows. Currently only format 1 is supported.
gid	Group ID this timing event belongs to.
id	The whole ID field of a timing event. Can be used instead of the subid fields (evtno, gid, etc).
reqnobeam	Flag allowing requesting to run this event without beam.
sid	ID of the sequence this timing event belongs to.
vacc	Virtual accelerator descriptor field.



Director Command Attributes

dest Name of the destination node, equivalent to target of defdst edge in cmdule..

destbeamproc Name of the destination beamprocess. Equivalent to dest to the beamproccess' entry node..

destpattern Name of the destination pattern. Equivalent to dest to the patterns eentry node..

pattern See Target Block.

permanent If true, the flow command will permanently change the default destination of the target block, else the change lasts only during the execution of the command..

prio Chooses the queue priority level this command will be written to at the target block (0 (lo), 1(mid), 2(hi)).

qhi Flush queue selection, flushes mid priority queue if true.

qil Flush queue selection, flushes high priority queue if true.

qlo Flush queue selection, flushes low priority queue if true.

qty Repetition quantity of a command. The sent command will be executed until qty reaches 0.

target Name of the destination node, equivalent to target of defdst edge in cmdule..

tvalid Time after which this command will be valid (executable), can be absolute or relative depending on vabs flag.

twait Timespan a wait command is going to wait. Can absolute or relative to current time, depending on flag.

vabs Chooses whether tvalid is interpreted as absolute value or an offset to current time sum..

Acronyms

ACM Association for Computing Machinery.
ADC Analogue-to-Digital Converter.
ADEV Allan Deviation.
AES Advanced Encryption Standard.
ARP Address Resolution Protocol.
ASIC Application Specific Integrated Circuit.
B2B Bunch to Bucket transfer.
BP Beam Process.
BPC Beam Production Chain.
BTB Bunch-to-Bucket.
CB CrossBar.
CERN European Centre for Nuclear Research.
CMD-Q Command Queue.
CNF Conjunctive Normal Form.
CORBA Common Object Request Broker Architecture.
CPLD Complex Programmable Logic Device.
CPU Central Processing Unit.
CRC Cyclic Redundancy Check.
CS Control System.
DAC Digital-to-Analogue Converter.
DC Direct Current.
DDS Direct Digital Synthesis.
DESY Deutsches Elektronen Synchrotron (German Electron Synchrotron).
DHCP Dynamic Host Configuration Protocol.
DM Data Master.
DNF Disjunctive Normal Form.
DPRAM Dual Port RAM.
DSP Digital Signal Processor.
EB Etherbone.

EBM	Etherbone Master.
EBS	Etherbone Slave.
ECA	Event Condition Action unit.
EDF	Earliest Deadline First.
EPICS	Experimental Physics and Industrial Control System.
ESR	Experimentier-Speicherring.
Eth	IEEE 802.3 Ethernet.
FAIR	Facility for Antiproton and Ion Research.
FEC	Forward Error Correction.
FEC	Frontend Controller.
FESA	Frontend Software Architecture.
FH	University of Applied Sciences.
FIFO	First In, First Out.
FOI	Flow Of Interest.
FPGA	Field Programmable Gate Array.
FSM	Finite State Machine.
GbE	Gigabit Ethernet.
GMT	Greenwich Mean Time.
GMT	General Machine Timing.
GPS	Global Positioning System.
GSI	GSI-Helmholtz Centre for Heavy Ion Research.
HDL	Hardware Description Language.
HP	High Priority.
HW	hardware.
IEEE	Institute of Electrical and Electronics Engineers.
IETF	Internet Engineering Task Force.
IO	Input/Output.
IP	Internet Protocol Version 4.
IRIG	Inter Range Instrumentation Group.
ITU	International Telecommunication Union.
JTAG	Joint Test Action Group.
LHC	Large Hadron Collider.
LINAC	Linear Accelerator.
LM32	Lattice Mico 32 Processor.
lo	left over service.
LP	Low Priority.
LSA	LHC Software Architecure.
LVDS	Low Voltage Differential Signalling.
MCU	Microcontroller Unit.
MIL	MIL-STD-1553.
MSI	Message Signalled Interrupt.

MTS Minimal Test System.
MUX Multiplexer.
NC Network Calculus.
NCO Numerically Controlled Oscillator.
NIC Network Integrated Controller.
NTP Network Timing Protocol.
NW Network.
OSI Open Systems Interconnection Model.
PAL Programmable Array Logic.
PBOO Pay Burst Only Once.
PCIe Peripheral Component Interconnect express.
PGS Packetised Greedy Shaper.
PHY Physical interface.
PLA Programmable Logic Array.
PLC Programmable Logic Controller.
PLL Phase Locked Loop.
PMOO Pay Multiplexing Only Once (Analysis).
PPS Pulse Per Second.
PQ Priority Queue.
PSCED Packetised Scheduler, Earliest Deadline First.
PTP Precision Time Protocol.
QoS Quality of Service.
RAM Random Access Memory.
RC Resistor-Capacitor.
RDMA Remote Direct Memory Access.
RF Radio Frequency.
RR Round Robin.
RT real-time.
RTC Realtime Clock.
RTOS Realtime Operating System.
RTS Realtime System.
RX Receiver.
SCPU Soft Central Processing Unit.
SDB Self Describing Bus.
SDH Synchronous Digital Hierarchy.
SERDES SERialiser / DESerialiser.
SFA Separate Flow Analysis.
SI International System of Units.
SIS100 Schwerionen Synchrotron 100.
SIS18 Schwerionen Synchrotron 18.
SIS300 Schwerionen Synchrotron 300.

SNMP Simple Network Management Protocol.
SOAP Simple Object Access Protocol.
SoC System on a Chip.
SONET Synchronous Optical Networking.
SSL Secure Socket Layer.
ST Standard Time.
SW software.
SyncE Synchronous Ethernet.
TAI International Atomic Time.
TCP Transmission Control Protocol.
TDC Time to Digital Converter.
TFA Total Flow Analysis.
TLU Timestamp Latch Unit.
TR Timing Receiver.
TS Timestamp.
TTF Timing Test Facility.
TX Transceiver.
UDP User Datagram Protocol.
USB Universal Serial Bus.
UT Universal Time.
UTC Coordinated Universal Time.
VAT Virtual Address Table.
VC(X)O Voltage Controlled (Crystal) Oscillator.
VME Versa Module Eurocard-bus.
WB Wishbone.
WBM Wishbone Slave.
WBS Wishbone Master.
WR White Rabbit.

Table 8.1: Schedule – Applicable attributes per node type

Attribute	Schedule Node Type						
	tmsg	noop	flow	flush	wait	block	blockalign
cpu	✓	✓	✓	✓	✓	✓	✓
thread	✓	✓	✓	✓	✓	✓	✓
patentry	✓	✓	✓	✓	✓	✓	✓
patexit	–	–	–	–	–	✓	✓
pattern	✓	✓	✓	✓	✓	✓	✓
bpentry	✓	✓	✓	✓	✓	✓	✓
bpexit	–	–	–	–	–	✓	✓
beamproc	✓	✓	✓	✓	✓	✓	✓
tperiod	–	–	–	–	–	✓	✓
qil	–	–	✓	–	–	✓	–
qhi	–	–	✓	–	–	✓	–
qlo	–	–	✓	–	–	✓	–
toffs	✓	✓	✓	✓	✓	–	–
id	✓	–	–	–	–	–	–
tvalid	–	✓	✓	✓	✓	–	–
vabs	✓	✓	✓	✓	–	–	–
prio	✓	✓	✓	✓	–	–	–
qty	✓	✓	–	–	–	–	–
twait	–	–	–	✓	–	–	–
permanent	–	–	✓	✓	–	–	–

Table 8.2: Command – Applicable attributes per node type

Attribute	Cmd Node Type I					
	start	stop	noop	flow	flush	wait
cpu	✓	✓	✓	✓	✓	✓
thread	✓	✓	✓	✓	✓	✓
target	✓	✓	✓	✓	✓	✓
pattern	✓	✓	✓	✓	✓	✓
beamproc	✓	✓	✓	✓	✓	✓
dest	–	–	–	✓	✓	–
destpattern	–	–	–	✓	✓	–
destbeamproc	–	–	–	✓	✓	–
qil	–	–	–	–	✓	–
qhi	–	–	–	–	✓	–
qlo	–	–	–	–	✓	–
tvalid	–	–	✓	✓	✓	✓
vabs	–	–	✓	✓	✓	✓
prio	–	–	✓	✓	✓	✓
qty	–	–	✓	–	–	–
twait	–	–	–	–	✓	–
permanent	–	–	–	✓	✓	–

Table 8.3: Command – Applicable attributes per node type (continued)

Attribute	Cmd Node Type II			
	staticflush	lock	unlock	asyncclear
cpu	✓	✓	✓	✓
thread	✓	✓	✓	✓
target	✓	✓	✓	✓
pattern	✓	✓	✓	✓
beamproc	✓	✓	✓	✓
dest	✓	–	–	–
destpattern	✓	–	–	–
destbeamproc	✓	–	–	–
qil	✓	–	–	–
qhi	✓	–	–	–
qlo	✓	–	–	–
tvalid	–	–	–	–
vabs	–	–	–	–
prio	–	–	–	–
qty	–	–	–	–
twait	–	–	–	–
permanent	–	–	–	–

Table 8.4: Schedule – Valid edge types per node type

Edge Type	Node Type								
	tmsg	noop	flow	flush	wait	lock	unlock	asyncclear	block	blockalign
defdst	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
altdst	–	–	–	–	–	–	–	–	✓	✓
target	–	✓	✓	✓	✓	✓	✓	✓	–	–
flowdst	–	–	✓	–	–	–	–	–	–	–
flushovr	–	–	–	✓	–	–	–	–	–	–
dynid x	✓	–	–	–	–	–	–	–	–	–
dynpar0	✓	–	–	–	–	–	–	–	–	–
dynpar1	✓	–	–	–	–	–	–	–	–	–
dyntef	✓	–	–	–	–	–	–	–	–	–
dynres	✓	–	–	–	–	–	–	–	–	–

Chapter 9

Troubleshooting

(ToDo)

Appendix I

Attached Documents

None.

Appendix II

Document Information

II.1 Document History

Version	Date	Description	Author	Review / Approval
0.1.0	01. Feb. 2018	created	M. Kreider	Pending