WEIGHTED SCHEDULING IN HETEROGENEOUS ARCHITECTURES FOR OFFLOADING VARIABLE-LENGTH KERNELS

by

PRATIKAKIS MENELAOS

B.A. CSD, University of Crete, 1998

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

DEPARTMENT OF ELECTRONIC ENGINEERING

SCHOOL OF APPLIED SCIENCES

TECHNOLOGICAL EDUCATIONAL INSTITUTE OF CRETE

2015

Approved by:

Major Professor
Kornaros George

Abstract

Heterogeneous System Architecture (HSA) is a type of computer processor architecture that integrates different processor architectures, for example central processing units and graphics processors, on the same bus with shared tasking and memory. These systems have different processes from different sources, with different priorities and weights, which are required to be performed by different processors architectures.

The above is achieved by scheduling. Scheduling is the process by which processes are given access to system resources (e.g. processor cycles, communications bandwidth). The demand for fast computer systems, the execution of multiple processes simultaneously (multitasking) and requirement for transmitting multiple flows simultaneously (multiplexing) have as a result the need for an efficient scheduling algorithm. The basic function of the scheduler is to determine which process will be run when there are several runnable processes. Therefore the scheduler choices have an impact on the utilization of system resources and other performance parameters. There exists a number of CPU scheduling algorithms like First Come First Serve, Shortest Job First Scheduling, Round Robin scheduling, Priority Scheduling etc, but due to a number of disadvantages these are rarely used in real time operating systems except Round Robin scheduling. Especially in a heterogeneous multicore system with existence of multiple queues with different priority and weight each, the scheduling/ dispatching of each queue separately and on the whole, is a critical issue. The purpose is to find, study and implement in a program language such us C, an algorithm to achieve a better management in such queues.

Σύνοψη

Σε ένα ετερογενές σύστημα αρχιτεκτονικής συνδυάζονται διαφορετικές αρχιτεκτονικές επεξεργαστών, για παράδειγμα, κεντρικών μονάδων επεξεργασίας και επεξεργαστές γραφικών, οι οποίοι μπορεί να συνδέονται στον ίδιο δίαυλο, να μοιράζονται διεργασίες και να έχουν κοινόχρηστη μνήμη. Τα συστήματα αυτά δέχονται διαφορετικές διαδικασίες από διάφορες πηγές, με διαφορετικές προτεραιότητες και βάρη, οι οποίες για την εκτέλεση τους απαιτούν διαφορετικές αρχιτεκτονικές επεξεργαστών.

Τα παραπάνω επιτυγχάνονται με την χρονοδρομολόγηση. Η χρονοδρομολόγηση είναι η διαδικασία με την οποία οι διεργασίες αποκτούν πρόσβαση στους πόρους του συστήματος (π.χ. επεξεργαστή, μνήμη κ.α). Η ανάγκη για έναν αλγόριθμο χρονοδρομολόγησης προκύπτει από την απαίτηση γρήγορων υπολογιστών συστημάτων για την επίτευξη πολυεπεξεργασίας (εκτέλεση περισσότερων από μία διεργασία κάθε φορά) και πολυπλεξίας (ταυτόχρονη μετάδοση πολλαπλών ροών). Η χρονοδρομολόγηση είναι μια θεμελιώδης λειτουργία του λειτουργικού συστήματος που καθορίζει ποια διαδικασία θα εκτελεστεί, όταν υπάρχουν πολλές εκτελέσιμες διαδικασίες.

Ο τρόπος χρονοδρομολόγησης της CPU είναι ιδιαίτερα σημαντικός επειδή έχει αντίκτυπο στην αξιοποίηση των πόρων του συστήματος και στις παραμέτρους των επιδόσεων. Υπάρχει μια πληθώρα από αλγόριθμους χρονοδρομολόγησης όπως η ουρά προτεραιότητας, η συντομότερη εργασία πρώτη, η χρονοδρομολόγηση Round Robin, η χρονοδρομολόγηση με βάση την προτεραιότητα κλπ, αλλά εξαιτίας μιας σειράς από μειονεκτήματα αυτές οι τεχνικές σπάνια χρησιμοποιούνται στα λειτουργικά συστήματα πραγματικού χρόνου, εκτός της χρονοδρομολόγησης Round Robin. Ειδικά σε ένα ετερογενές σύστημα πολλαπλών πυρήνων, με την ύπαρξη πολλαπλών ουρών, με διαφορετική προτεραιότητα και βάρος η καθεμία, η διαδικασία χρονοδρομολόγησης/αποστολής διεργασιών από κάθε ουρά ξεχωριστά αλλά στο σύνολό τους, είναι ένα κρίσιμο ζήτημα.

Ο σκοπός της παρούσας εργασίας ήταν να βρεθεί, μελετηθεί και υλοποιηθεί σε μια γλώσσα προγραμματισμού, όπως η C, ένας αλγόριθμος, βασισμένος στα βάρη των εργασιών, για να επιτευχθεί καλύτερη διαχείριση τέτοιων ουρών.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my sincere gratitude to Dr. George Kornaros, Lecturer at the Department of Informatics Engineering of the Technological Educational Institute of Crete and supervisor of the present master thesis, for the continuous support of my Msc study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Msc study.

I would like to thank my family for supporting me spiritually throughout writing this thesis as well as my good friend Fr Dimitrios.

October 2015,

Menelaos Pratikakis

1 Introduction

Heterogeneous System Architecture (HSA) is a computer processor architecture that integrates central processing units and graphics processors on the same bus, with shared memory and tasks. The GPU has great processing power and the overwhelming majority of applications and computing tasks exploit the processing power offered by GPUs, therefore the HSA aims to properly utilize the processing power offered by GPUs. The HSA has developed from the HSA Foundation, whose founding members are AMD, ARM, MediaTek, Qualcomm, Texas Instrument, Imagination and Samsung [1]. HSA is widely used in system-on-chip devices, such as tablets, Smartphones and other mobile devices.

Scheduling is an increasingly important topic in HSA systems. Scheduling is the process by which processes are given access to system resources (e.g. processor cycles, communications bandwidth). The demand for fast computer systems, the execution of multiple processes simultaneously (multitasking) and requirement for transmitting multiple flows simultaneously (multiplexing) have as a result the need for an efficient scheduling algorithm. The basic function of the scheduler is to determine which process will be run when there are several runnable processes. Therefore the scheduler choices have an impact on the utilization of system resources and other performance parameters. There exists a number of CPU scheduling algorithms like First Come First Serve, Shortest Job First Scheduling, Round Robin scheduling, Priority Scheduling etc, but due to a number of disadvantages these are rarely used in real time operating systems except Round Robin scheduling [2].

1.1 Research questions and methodology

The purpose of the present thesis is the study of the following: a) How a scheduling algorithm for dispatching jobs to accelerator processors in a heterogeneous system can be implemented. b) What the behavior of the algorithms on various conditions of executions is. The structure of the remaining of this thesis is as follows. Firstly the objective of this work is presented. Next heterogeneous architecture, the architecture of the GPU and the programming environments of GPU are briefly presented. Afterwards the principles and criteria of scheduling are analyzed. Sequentially a description of basic scheduling algorithms is presented. Then the basic idea of our algorithm is analyzed and the way of

implementation is described. Next measurements based on specific scenarios are presented. The last part includes the conclusions of the present work and future proposals.

2 Objective of the Study

The basic idea of the algorithm is as follows. In the context of a multiprogrammed environment we assume we have *N* applications, a set N_Q of queues with a fixed max weight (*weight_queue)* that the system assigns to each queue. The weight of each queue represents the maximum time quantum allocated to this queue for execution, or the normalized maximum time quantum. Each application enqueues jobs in queues. The jobs are dispatched to a hardware accelerator either single or multi-threaded (e.g. GPU). The queue contains packets with the job attributes, such as pointers to the kernel code and data, estimated execution time, type, etc.

The centralized scheduler when deals with a queue with a higher weight (*weight_queue*) compared to a queue with a less weight, must serve a number of packets in proportion to the ratio of their weights. The algorithm cannot service packets with total weight greater than the weight of the queue (*weight_queue)*. Additionally the total serviced weight of all queues should be less or equal than the maximum total weight of the algorithm, which is equal to the sum of all queues weights. When the total maximum weight of all queues weights has been achieved then a cycle is completed. Hence dequeuing packets must be enqueued before the cycle starts. If the packets have been enqueued during the new cycle, they will be served in the next cycles. Essentially, the algorithm scheduling policy works in Round-Robin fashion and provides weighted fairness for variable-length packets (i.e., execution time) maintained in multiple queues.

3 Heterogeneous architecture

The progress in semiconductor technology has brought evolving microprocessors developed for a wide range of applications such as aerospace, power electronics, defense systems, geosciences, bioinformatics, interactive digital media, cloud computing, etc. In heterogeneous multi-core systems for specialized purposes, the cores are integrated into the same chip specific processor/functional units and general-purpose cores [3]. Heterogeneous computing refers to systems that use more than one kind of processor. Therefore, the opportunity to accelerate emergency applications by running critical tasks on fast cores is given. This embodiment has advantages in areas such as performance, power optimization. Especially in the last decade, multi-core processors are increasingly used because of the high performance they provide, while they have reduced their energy requirements.

A heterogeneous computer cluster is more effective than a homogeneous since some types of processing units perform better than others in certain processing tasks. Furthermore the closely tied hardware accelerator within a node can reduce communication requirements by making use of locality data. Overall system performance can be improved by allowing the heterogeneous cores to work collaboratively on different parts of an application.

Therefore commercial operating systems have been improved to support the parallelism offered by multi-core processors. Furthermore, the need for extensive battery life in portable devices and high performance, has led to power/efficient performance and ultra low power small cores (e.g. Intel's Atom processor). Since available different types of cores, architectural options when designing a platform are also more. The possibility of developing heterogeneous architectures, combining large and small cores on the same die, in order to provide a range of power / performance capacity is also given. In addition to the large and small cores, on-die integration in specific areas accelerators for operating special purpose, such as graphics and media processing has become widespread.

According to Kyriazis G [4], the essence of the HSA strategy is to create a single unified programming platform providing a strong foundation for the development of

languages, frameworks, and applications that exploit parallelism. More especially, HSA's objectives include:

• The use of the processing power offered by GPUs

• Removing the programming dam between CPU/GPU.

• Reduced CPU / GPU latency communication status.

• The opening of the programming platform to a wider range of applications by enabling existing programming models.

• Create a base for registration of additional processing elements beyond the GPU and CPU.

An HSA application is run on a various platforms comprising both CPUs and Intellectual Property (IPs) such GPUs. HSA permits the application to execute at the best possible performance and power points on a certain platform, without dispensing flexibility. Simultaneously, it improves programmability, portability and compatibility.

Figure 3-1 indicates a simple HSA platform. The HSA Accelerated Processing Unit (APU) includes a GPU with multiple HSA compute units (H-CUs), a multicore CPU, and the HSA memory management unit (HMMU). The above components are in communication with coherent and non-coherent system memory.

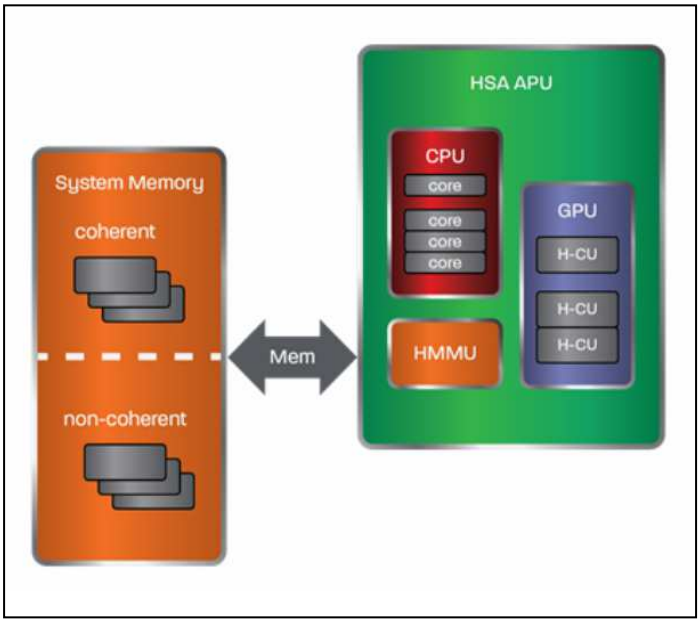

Figure 3-1 A simple HSA platform [4]

3.1 Types of heterogeneous architecture

For heterogeneous systems factors such as performance, power, flexibility and programmability should be taken into account. According to Chitlur, N. *et al* [4], types of heterogeneous architecture configuration can be described as follows:

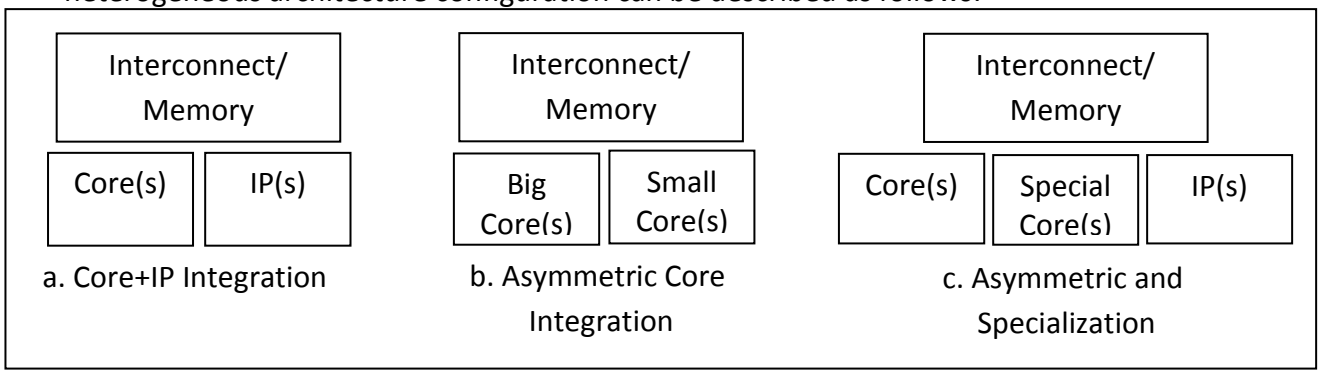

Figure 3.1-1 : Heterogeneous Architectures Under Exploration [5]

Core+IP Integration[5]: This type of architecture (illustrated in Figure 1) integrates multiple homogeneous cores with hardware accelerators (also known as intellectual property (IP)). In this type of architecture, the IP block is low power but achieves high-performance process for specific areas such as graphics, security, imaging, etc.

System on Chip	CPU	GPU	Devices
Exynos 5 Dual	1.7 GHz dual-core ARM Cortex-A15	ARM Mali-T604 (quad-core)	SamsungChromebookXE303C12,[Google Nexus 10,
Tegra 3 T30L	1.2 GHz quad core (up to 1.3 GHz in single-core mode)	12 core	Lenovo IdeaPad Yoga 11, Acer Iconia Tab A700, ZTE Era,
Teggra 4 T114	Up 1.9 GHZ quad core Cortex-A15	72 cores	Tegra Note 7, Microsoft Surface 2, HP SlateBook x2, Toshiba AT10-LE-A
Tegra K1 T132	up to 2.5 GHZ dual core Denver (64bit)	192 core	Google Project Tango tablet,
Texas Instument OMAP4460	1.2–1.5 GHZ, dual core Cortex -A9	PowerVR SGX54	Samsung Galaxy Nexus, Archos 80 Turbo, Huawei Ascend D1

Table 3.1-1 Examples of Core+IP Integration [6]

Asymmetric Core Integration[5]: This type of heterogeneous architecture is proposed by ARM Holdings and combines a number of general purpose cores. The cores are asymmetric

in power consumption and performance. Therefore large and small cores are collaborating to provide the power efficiency or performance when needed, with the probability of many same large and small pairs on a chip. The cores could be of different generations, although they are usually from the same ISA family.

Each pair (large and small cores) is considered as a virtual core. In real time only one core is active and running at any time. Hence the large core is used when the system requirements are high, whereas if the system requirements are low used the small core is used. When request for virtual core is alternating between low and high, the incoming core is enabled, the operation state is transferred, the outgoing core is closed down, and processing continues on the new core. [6]

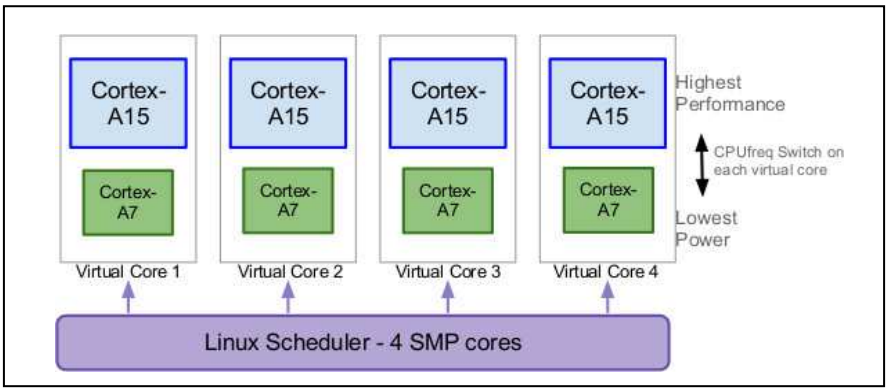

Figure 3.1-2 CPU migration via the in-kernel switcher [6]

The most powerful model of small and large cores is heterogeneous multi-processing (MP). This type allows the simultaneous operation of all the cores regardless of size. So processes with large computational requirements or high priority are executed by large cores. On the other hand, processes with less computational requirements or less priority can be performed by the small cores. [6]

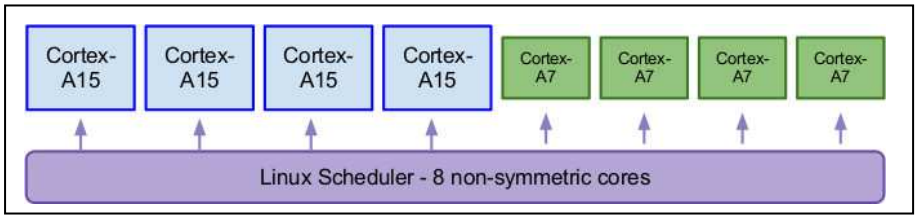

Figure 3.1-3 heterogeneous multi-processing (MP)[6]

System on Chip	big cores	LITTLE cores	GPU	Devices
HiSilicon K3V3	1.8 GHz dual-core Cortex-A15	1.2 GHz dual-core Cortex-A7	Mali-T658	
HiSilicon Kirin 920	1.7-2.0 GHz Cortex-A15	1.3-1.6 GHz quad-core Cortex-A7	Mali-T628MP4	Huawei Honor 6
HiSilicon Kirin 930	Cortex-A53 quad core 2.0 GHz	1,5 Ghz quad core Cortex-A53	Mali-T628 MP4	Huawei P8
Samsung Exynos 5 Octa	1.6-1.8 GHzquad-coreCortex-A15	1.2 GHz quad-core Cortex-A7	PowerVR SGX544MP3	Exynos 5-basedSamsung Galaxy S4, ZTE Grand S II
Samsung Exynos 5 Octa	1.8-2.0 GHz quad-core Cortex-A15	1.3 GHz quad-core Cortex-A7	Mali-T628MP6	Exynos 5-basedSamsung Galaxy Note 3, Samsung Galaxy Tab Pro, Galaxy S5 SM-G900H
Samsung Exynos 5 Octa	2.1 GHz quad-core Cortex-A15	1.5 GHz quad-core Cortex-A7	Mali-T628MP6	Exynos 5-basedSamsung Galaxy S5-G900, Odroid-XU3, Odroid-XU4
Samsung Exynos 5 Hexa	1.7 GHz dual-core Cortex-A15	1.3 GHz quad-core Cortex-A7	Mali-T624	Samsung Galaxy Note 3 Neo
Samsung Exynos 5 Octa	1.8 GHz quad-core Cortex-A15	1.3 GHz quad-core Cortex-A7	Mali-T628MP6	Samsung Galaxy Alpha
Samsung Exynos 7 Octa	1.9 GHz quad-core Cortex-A57	1.3 GHz quad-core Cortex-A53	Mali-T760MP6	Samsung Galaxy Note 4 (SM-N910C)
Samsung Exynos 7 Octa (7420 model)	2.1 GHz quad-core Cortex-A57	1.5 GHz quad-core Cortex-A53	Mali-T760MP8	Samsung Galaxy S6, Samsung Galaxy S6 Edge
Renesas Mobile MP6530[	2 GHz dual-core Cortex-A15	1 GHz dual-core Cortex-A7	PowerVR SGX544	
Allwinner A80 Octa	Quad-core Cortex-A15	Quad-core Cortex-A7	PowerVRG6230	

MediaTekMT6595	2.2 GHz quad-core Cortex-A17	1.7 GHz quad-core Cortex-A7	PowerVR G6200 (600 MHz)	
MediaTek MT6595M	2.0 GHz quad-core Cortex-A17	1.5 GHz quad-core Cortex-A7	PowerVR G6200 (450 MHz)	
MediaTek MT6595 Turbo	2.5 GHz quad-core Cortex-A17	1.7 GHz quad-core Cortex-A7	PowerVR G6200 (600 MHz)	
QualcommSnapdragon 808 (MSM8992)	2.0 GHz dual-core Cortex-A57	Quad-core ARM Cortex-A53	Adreno 418	LG G4
Qualcomm Snapdragon 810 (MSM8994)	2.0 GHz quad-core Cortex-A57	Quad-core ARM Cortex-A53	Adreno 430	HTC One M9, LG G Flex 2, OnePlus 2
Nvidia Tegra4 T40	1.9 GHz quad-core ARM Cortex-A15]+	1 low power core	Nvidia GeForce @ 72 core	Nvidia ShieldTegra Note 7
Nvidia Tegra4 AP40	1.2-1.8 GHz quad-core	1 low power core	Nvidia GPU 60 cores	

Table 3.1-2 Examples of big-LITTLE heterogeneous multi-soc[6]

Asymmetry+Specialization[5]: The third type of configuration combines asymmetric cores, special purpose cores and hardware accelerators**.** The main difference lies in the special purpose cores which are used for special aims (hardware scheduling, management, etc).

3.2 GPU architecture

A number of the multi-cores architectures have developed to meet the needs for processing power. GPU architecture is one of the most powerful. GPU architectures have multiple intensive processors that are specialized for the execution SIMT (Single Instruction Multiple Thread) operating activities. Until now, the performance of GPU architecture is at least six times faster than the general purpose CPU architecture [7]. The computer scientists were particularly interested in exploiting this computing power to quickly solve large general purpose problems, known as General-Purpose computing on the GPU (GPGPU), utilizing the potential of parallel programming. A platform called general-purpose computing on graphics processing units (GPGPU) has emerged to optimize the performance of GPU. GPGPU programs usually consist of two parts: kernel code and host code. Kernel code is executed on multiple GPU cores with the configuration. The host code is executed in

CPU and includes mainly the procedure for preparing the GPU device, data transfer between the GPU and the host, as well as the launching of kernels with configuration [8].

Modern GPUs consist of hundreds of processing units operating at low to medium frequency, designed for throughput -oriented latency insensitive workload. To hide global memory latency, GPUs contain small or moderate sized on-chip caches, and make wide use of hardware multithreading, performing tens of thousands of threads simultaneously throughout the pool of processing units. The GPU processing units are usually organized in single-instruction multiple-data (SIMD) clusters controlled by a single instruction decoder, with access to fast on-chip caches and shared memories. The SIMD clusters execute instructions in lock-step and branch divergence treated with the implementation of both paths of the branch and concealing results from inactive processing units as necessary. The use of SIMD architecture and in-order execution of instructions permits GPUs to contain a greater number of arithmetic units in the same area as compared to conventional CPUs [9].

Due to the high computational requirement of graphics GPUs achieved single-precision floating point arithmetic rates approaching 2 trillions of instructions per second. GPUs are designed with global memory systems capable of bandwidths approaching 200 GB/sec. GPU memory is organized into multiple banks. Maximum performance is achieved when the accesses are aligned with the appropriate address boundaries. When a memory access is not aligned with an appropriate address boundary and in consecutive sequence, the memory access must be divided into multiple transactions resulting in a significant decrease of the effective bandwidth and increasing latency [9].

Even though the GPU are powerful computing modules in their own right, their management is done by host CPU. The GPUs are usually connected with the host through PCI-Express bus and in most cases they have their own independent memory. To achieve data exchange with GPU, the host CPU performs DMA transfers between GPU memory systems and the host, and in some cases, allow their on-board memory to be mapped in the host's address space, therefore data is read or written only once during kernel execution [9].

3.2.1 CUDA-OPEN CL

For programming GPU’s, various programming environments have been developed. GPUs programming models initially consisted of specialized high-level programming

languages, such as HLSL, GLSL, and Cg [10]. In particular, after 2006, where NVIDIA opened its CUDA (Compute Unified Device Architecture) architecture, it eliminates the need to use the graphics application programming interfaces (API) for calculating applications, allowing utilization of GPU computing to more widespread use. Additionally, the Advanced Parallel Processing (APP) which enables API's GPUs, working together with the CPU, is combined with the ability of programmers to develop GPU computing application without mastering graphics terms. At the same time it enabled the acceleration of the execution of applications and makes the coding of large programs easier.

The two modern programming GPU interfaces are CUDA and Open Computing Language (OpenCL). OpenCL, a portable language for GPU, is an open standard maintained by the non-profit technology consortium Khronos Group. CUDA is a C language framework. It is specifically for NVIDIA GPUs with set of language extensions that works only on NVIDIA's GPUs, while OpenCL is an open standard that can be used to program central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors.

CUDA and Open/CL are quite similar to each other; they have similar programming models, execution models, memory models and platform models but different programming interfaces. For a programmer, the computing system consists of a host (a typical CPU), and one or more devices providing parallel processors and a large number of arithmetic execution units. Furthermore their built-in functions and syntax for various keywords are similar. Thus it is relatively easy to translate a CUDA program in an Open/CL program [10, 11].

4 Background on scheduling algorithms

Theorem: Dertouzos and Mok say: "*No scheduling algorithm is optimal for scheduling hard real-time aperiodic tasks on two or more processors if all release times, deadlines, and execution requirements are not known a priori*" as mentioned by Srinivasan A. [12]

4.1 General principles of Scheduling Algorithms

In CPU scheduling we accept the following assumptions. There are a number of runnable processes waiting for the CPU. Waiting is performed in an "area" called job pool. Also all processes compete for resources and independent of each other. The main job of the scheduler is to distribute the CPU resources fairly and in a way that optimizes certain performance criteria

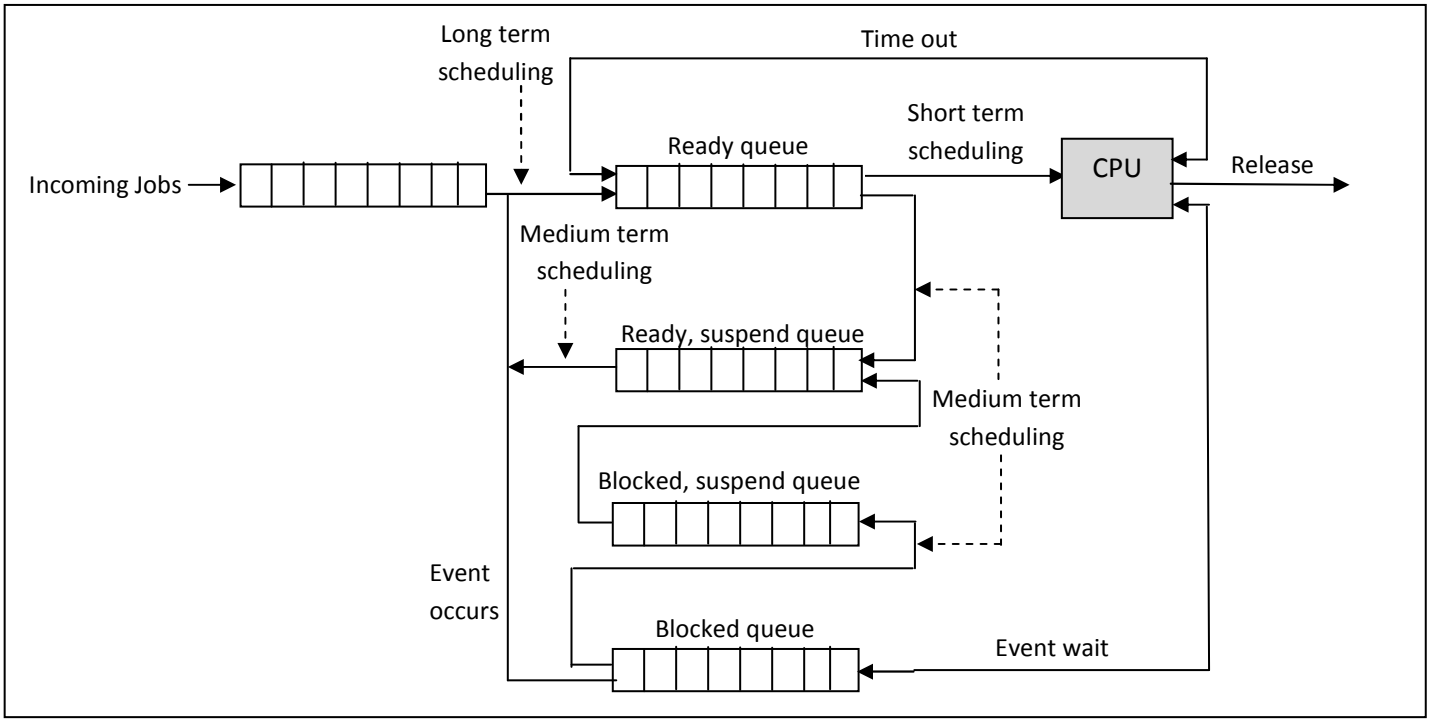

Figure 4.1-1 Queuing diagram for scheduling [2]

From the ready queue, the short term scheduler, known as CPU scheduler, select a process to be executed and allocates the CPU. The scope of medium term scheduler is to remove processes from memory and to reduce the degree of multiprogramming results in

swap system. Swap is performed by a dispatcher. A dispatcher is the unit that gives control of the CPU to the process.

Especially in real-time systems, where there are time restrictions in the calculations, the CPU scheduler performs an important role. In such a system the processes has to be completed within specified time restrictions. Most real-time systems can be applied in unpredictable environments that can handle unknown and changing tasks. Therefore a dynamic task scheduling is necessary. Additional software and system hardware must adapt to unforeseen compositions.

There are two main types of real-time systems [13]: Hard Real-Time System, and Firm or Soft Real-Time System. In Hard Real-Time System specified deadlines must be complied. Otherwise the result could be disastrous. Soft Real-Time System has higher tolerance. Such systems where performance is limited, but where are no catastrophic consequences in case of failure to meet the time constraints, are called soft real time systems. In real-time systems each task must be completed before its deadline. In soft real-time system the simple Round Robin algorithm has as a result low throughput and as a consequence more number of context switches, longer response and waiting times. On the other hand, if such a system has a large CPU burst, this can lead to starvation problem. Priority scheduling may be a better choice in real-time systems, but still there is the problem of starvation, due to a low priority processes will forced to wait.

4.2 Scheduling Criteria

The basic Scheduling Criteria are [13]:

- *CPU Utilization* - how busy the CPU is.
- *Context Switch*: It is the process of storing and retrieving the state of a non-integrated process, so that the process can be executed later, starting from the last saved context. It usually requires computing power, leads to memory waste and time, thus increasing the overhead of scheduler.
- *Throughput* – depends on the number of processes that are completed per unit time Throughput and context switching and are inversely proportional.

- *Turnaround Time*- How long it takes to execute a process. Turnaround time derived from the sum of the waiting times to get into memory, waiting time in the ready queue, the execution time for the CPU and time for the necessary I/O.
- *Waiting Time*- It is the sum of periods spent in ready queue and it is directly dependent on the scheduling algorithm.
- *Response Time*-. How long it takes until the first response after a process request.

For a scheduling algorithm to be optimally it must achieve maximum CPU utilization, maximum context switches and throughput, but minimum turnaround time, minimum waiting time and response time.

4.3 General Scheduling algorithms

Scheduling algorithms can be divided to static and dynamic algorithms. Static algorithms have fixed priorities assigned to classes and always prefer one class over another.

4.3.1 First Come First Served (FCFS)

An example of a static algorithm is First Come First Served (FCFS) algorithm. FCFS is the oldest and simplest scheduling algorithm. FCFS can be implemented using a First Come First Served (FIFO) queue. This implementation is simple with minimal overhead on CPU. FIFO has only one queue, and the packet that arrived first also gets sent out first. Packets are sent in the order in which they arrive without hierarchy. This can be done either by a linked list or a ring buffer, or a hash table indexed by the values of packet arrival time. The latter method is used in many devices based on specially designed integrated circuits, while the two former methods are more common in cores operating systems like Linux or BSD. FIFO is a natural choice where queues do not require any hierarchy.

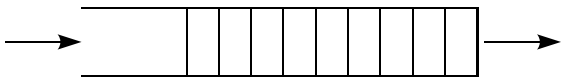

Figure 4.3.1-1 First Come First

Main Characteristics:

- There is no prioritization which means that each process may eventually be completed, therefore, no starvation.

- The process with the longer burst time can monopolize the CPU, even if the burst time of another process is very small. Therefore, yield is low. [14]
- The algorithm is seldom chosen since the process takes all resources until completion.
- Convoy effect [15] is a crucial issue. It occurs when more than one processes share the same resources. If a long process has reserved resources for a very long time, the new short processes that are scheduled cannot be served. As a result they can cause additional delays and significantly increase the system load.

4.3.2 Shortest Job First (SJB) [14]

The process assigned to the CPU has an execution time at least equal to the burst time. The scheduler sorts the processes according to the execution time. Processes with a short time burst are positioned in the starting of the queue and processes with longer burst time at the end of the queue. This algorithm requires an assessment of the integration time required for each process [14]. The design of this algorithm aims at maximum efficiency in most scenarios.

The main operating mode of the algorithm is as follows. The process having the shorter burst is allocated in CPU. If two processes have the same burst time, allocated first process came first according to the FCFS algorithm.

Main Characteristics:

- We must have knowledge of the length of the next CPU request. This is a problem with the SJF algorithm.
- The algorithm SJB reduced the average waiting time because it first served small process and then the larger processes.
- Although the average waiting time is reduced, it may adversely affect the processes with a long burst. In extreme cases, they will never serve the processes with large burst time, which is a major issue of this algorithm.
- Long running jobs may starve, low supply of short jobs to CPU.
- SJF is optimal in waiting time, by achieving minimum average waiting time.

4.3.3 Priority scheduling

Priority scheduling algorithms [15] is a basic classful scheduling algorithm in which each process has a value that represents the importance of task in the system. This value defines the priority. In the process with the highest priority, the resources are available for its completion. It consists of multiple classes with static priority [16]. It can be implemented either by each class having its own queue, ordinarily FIFO, either by a single sorted queue in with the higher priority tasks are at the front and the low priority at the back of the queue.

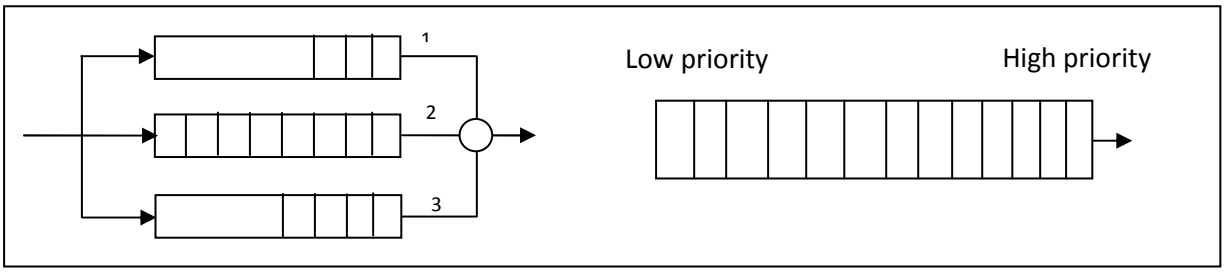

Figure 4.3.3-1 Priority queuing

Higher priority queue must be empty before selecting a task from a lower priority. This is the cause of starvation that the algorithm may suffer.

There are two techniques of priority scheduling algorithms, dynamic or static priority [17]. In dynamic algorithms, priority changes during execution, it either decreases or increases according to specific mechanisms.

Figure 4.3.3-2 shows a job with static priority (blue line), which has priority set to 250, and a task the priority of which decreases with time (red line).

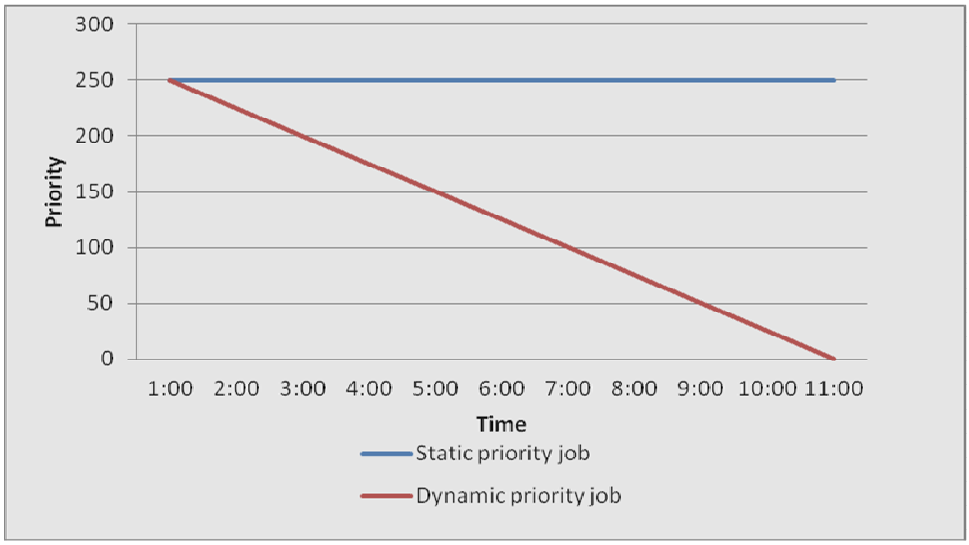

Figure 4.3.3-2 Static and dynamic priority

In static priority, the priority of a process never changes. It is defined at the start and remains the same until its completion. This kind of algorithms can suffer of job starvation. Whenever a process is ready for execution but there are no resources available, the process must wait. If, however, processes with higher priority arrive continuously, then the process which was waiting will never be served. This case is known as starvation [17].

One way of avoiding starvation is to determine the number of times that a process can be overcome by higher priority tasks. Another technique is called aging during which the priority of a process increases over time. At some point the process will succeed is getting the necessary resources. The worst is as the process with the highest priority to come up so it must be executed first. Nevertheless aging technique introduces computing overhead because of the calculation of new priorities. Another issue is when the appropriate determination of the time aging will take place. If this time is too short, then low priority tasks will turn into high priority tasks very quickly, thus loading to FCFS algorithm. On the other hand if the aging time is too long, maybe the technique will become partially ineffective.

4.3.4 Multilevel queue scheduling

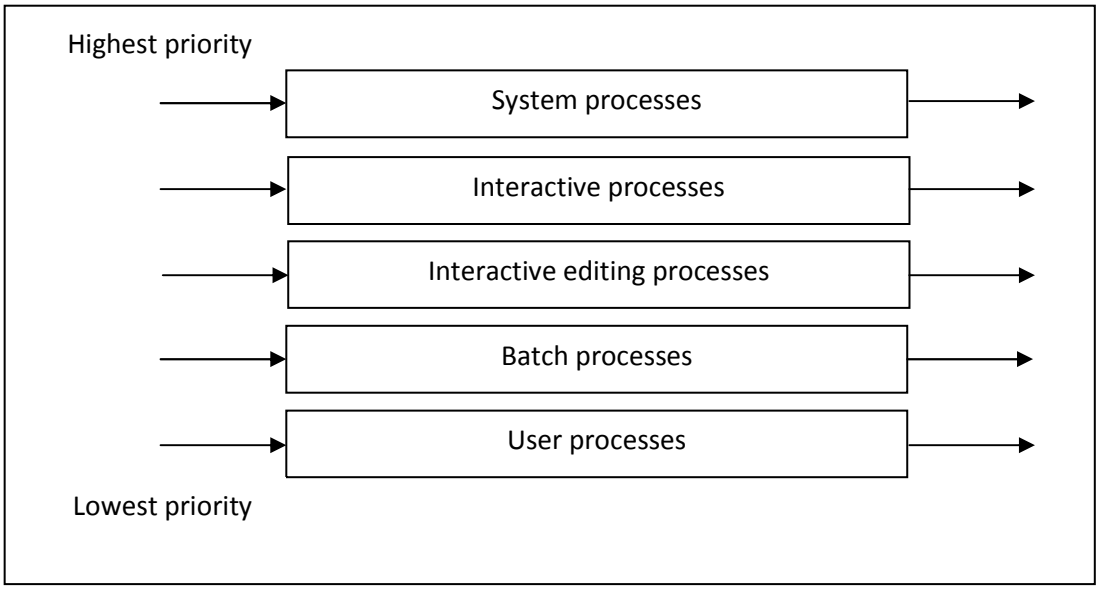

Figure 4.3.4-1 Multilevel queuing

The multilevel queue (MLQ) algorithm is based on the use of multiple queues. Each queue has different weight and the tasks are allocated to queues according to their importance. There is a queue for each category. Also for each queue a different algorithm

that will find the most important job of the queue can be selected. When resources are available the most important job is chosen from the queue with the highest priority to. If the queue is empty then the next queue is examined. If a task is found then resources are allocated and removed from the queue.

Main Characteristics:[15]

- Jobs cannot change queue. Therefore the right choice of queue is important for best results.
- Need to determine the number of queues.
- Determine the scheduling algorithm of each queue.
- The way by which it is decided which task will be placed in which queue.

4.3.5 Multilevel feedback queue scheduling

Multilevel feedback queue (MLFQ) [15] scheduling algorithms is an extension of MLQ algorithms. The main difference lies in the fact that the job can move from one queue to another queue. After a period of time quanta, the priority of process decreases and the priority queue changes. Also if a process is waiting in a queue too long this may increase the priority, and eventually it is transferred to another line with an increased priority. Nevertheless the main difficulty in applying the algorithm is MLFQ based on its complexity and because of usually introduced higher overhead.

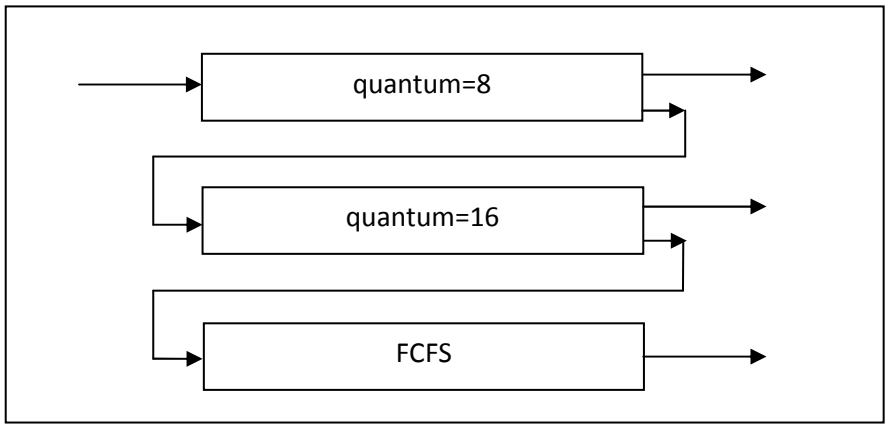

Figure 4.3.5-1 Multilevel feedback queuing

The problem of starvation is solved easily since it can change a job queue. Using parameters such as history and runtime information jobs can be distinguished at runtime according to their behavior. Problems with MLFQ algorithms can arise if a job changes behavior over time and the scheduler does not realize this change. In these cases the system

performance typically declines. Generally MLFQ algorithms achieve better results than other scheduling algorithms. However, they introduce overhead computational load. The implementation of Linux scheduler is based on MLFQ algorithm.

Main Characteristics

- the number of queues
- scheduling algorithms for each queue
- method used to determine when to upgrade a process
- method used to determine when to demote a process

4.3.6 Round Robin

The algorithm Round Robin (RR) [2, 14] assigns to each process a small unit of time, the time quantum. The schedule goes around this queue, allocating the CPU to each process for a specified time quantum. The new procedures are added to the end of the tail.

The Round Robin algorithm works as follows. At first, selected time is assigned to each process. Then the time is allocated to each process according to the FCFS algorithm. If the burst time is less than the quantum, then the carrying out of the process must be terminated. Otherwise the process is executed for as much time as quanta, and returns to the end of the queue waiting for the next cycle.

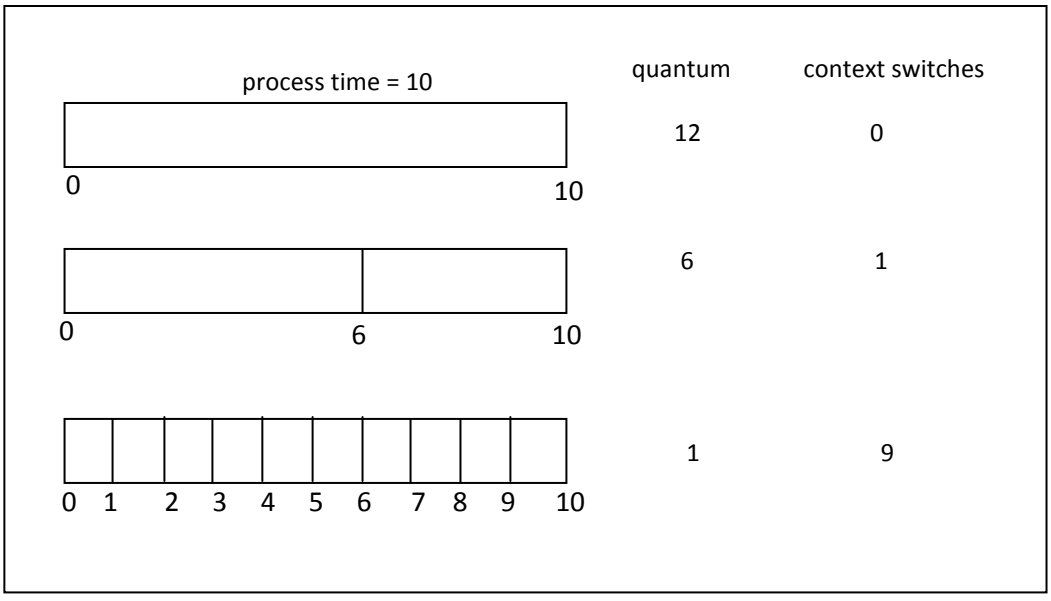

Figure 4.3.6-1 Context switches in Round Robin

Main Characteristics:

- The basic RR has large waiting time and large response time

- Therefore it provides less throughput.
- Can lead to reduced effectiveness of CPU, because of many context switches too short time quanta is selected.
- On the other hand, if the time quantum is too large, then the algorithm approximates the FCFS algorithm.
- Generally it achieves, higher average turnaround time than SJF, but better response.

4.3.7 Weighted Round Robin

Weighted Round Robin (WRR) is a scheduling algorithm with the main use in ATM networks using fixed size packets. It was first proposed by M. Katevenis, S. Sidiropoulos and C. Courcoubetis in 1991[18]. The basic idea is that a weight is assigned in each flow. In every cycle of service, the number of packets served is proportional to the weight defined for each flow. The job will receive w_i consecutive time slices in each round, and the duration of a round is the sum of all w_i.

Main Characteristics

- Equivalent to regular round robin if all weights are equal to 1.
- Simple to implement, since it doesn't require a sorted priority queue.
- Offers throughput guarantees - Each job makes a certain amount of progress each round.
- By giving each job a fixed fraction of the processor time, a round robin scheduler may delay the completion of every job.

4.3.8 Deficit Round Robin

Deficit Round Robin (DRR) is a queue scheduling algorithm based on round robin that is firstly applied to routing packets. It was proposed by M. Shreedhar and G. Varghese in 1995 as a fair and efficient, with O(1) complexity algorithm[19]. Patrick McHardy implemented DRR for Linux kernel

The main idea of DRR algorithm is as follow. There are queues of packets with specific flow for each queue. A quantum of service has been assigned to each queue and round robin technique is used to service the queues. The algorithm checks all the full tails in a sequence. If a non empty queue is found its deficit counter is incremented by its quantum value. The value of the deficit counter is a maximal amount of bytes that can be send at

each cycle. If the size of first packet of the queue is less than the deficit counter value, the packet can be sent and then the value of the deficit counter decreases by the packet size. Then, the size of the next packet is compared to the deficit counter value, etc. If the size of the first packet of the queue is greater than the deficit counter value, or the queue is empty, the scheduler will skip to the next queue; and the value of the deficit counter increases by the quantum value. If the queue is empty the deficit counter is set to zero. If a deficit counter becomes less or equal to zero then it increases by the quantum value.

Main Characteristics:

- The network administrator chooses weights of queues.
- Regardless of the size of each packet, it provides a minimum rate to each flow.
- If the quantum Q_i is larger than the maximum size of packet of each flow, the complexity of DRR is O(1).

4.4 Heterogeneous scheduled techniques

By default, GPU kernels executed serially, a kernel at a time. But the latest CUDA and Nvidia GPU architectures can perform multiple different kernels if resources are available. However the GPU kernels executes sequentially, if resources are insufficient. Sequential execution kernel could provide enough performance for most of the general-purpose computing. Nevertheless, in real-time systems, sequential execution can cause problems, because there is the potential for priority inversion. To resolve this problem numerous scheduled techniques have been proposed.

4.5 Heterogeneous Scheduling Categories

The main purpose of each scheduling algorithm is to assign a task to a suitable processor so that total execution time is minimized. However, to find a schedule for a heterogeneous parallel architecture must take into consideration a number of factors such as: different processing elements, processes may not be operable by all processors; the run time of a process may be different depending on processing elements and the communication time may vary [20].

Task scheduling on a heterogeneous system can be classified into several categories.

4.5.1 Static and dynamic Schedulers

One category is static and another one dynamic.

- In static, task-to-core mapping is done only once at the beginning of the application or at compile time offline. This model can be represented by dependency graph in which tasks in the critical path determine the total duration of application. This enables the application to accelerate by executing critical tasks in fast cores. Hence the schedule remains the same throughout execution of the application. Moreover all information about the tasks, such as the cost of execution and communication for each task and relationship with other tasks is known a priori [21].

  Static scheduling algorithms are classified into two categories, Heuristic-based and Guided Random Search-based algorithms. Heuristic-based algorithms often provide good solutions with polynomial time complexity. Guided algorithms Random Search-based also give approximate solutions [21].

- In dynamic scheduling, progress monitoring of works and migration are used to ensure a certain level of performance applications. Information about the tasks, as the cost of execution and communication cost for every task and the relationship with other tasks is not known beforehand. So decisions are taken at runtime [21]. The dynamic scheduling enables automatic synchronization and scheduling by the runtime system such us OmpSs [22]. This model maintains a directed acyclic graph (DAG) with the current state of the process, and when fulfilling the requirements of a process it becomes ready to be scheduled to an available core. Although the dynamic programming is likely to achieve better application performance than the static scheduling, yet it can be better in small multi -core systems and not in large multi-core systems [20].

4.5.2 Clustering, Listing, Duplication-based and Guided-random schedulers

Moreover schedulers for heterogeneous systems can be divided into four types of schedulers: clustering, listing, duplication-based, and guided-random schedulers [24, 25]. Clustering, listing and duplication-based algorithms belong to Heuristic-based class algorithm [21].

- Clustering schedulers comprise clusters and each cluster must be running on the same processor. Clustering heuristics are mainly proposed for homogeneous systems and they seem difficult to use in for heterogeneous systems. At the stage of clustering, the algorithm assumes that there are an infinite number of available processors. If the number of clusters is larger than the number of available cores, then it is necessary to merge the clusters so as to be as many as the available cores. Such type is the Levelized Min Time algorithm. With this method are organized clusters of tasks that can be performed in parallel depending on their level. Assuming a graph, sibling nodes in a graph have the same level. The priorities of tasks defined on a cluster according to the time of execution. Tasks with the largest execution time have the highest priority. The assignment task to the processor is in descending order of priority [23].
- Listing schedulers have two phases. In the first phase, every task is given priority based on the policy defined in each algorithm. In the second phase, according to their priorities, tasks are assigned to processors. These kinds produce the most efficient schedules, without compromising the makespan and with a quadratic complexity in relation to the number of tasks [21, 23].
- The aim of duplication-based schedulers is the limitation of communication between processes. This is achieved by duplication of tasks. If a task has a lot of successors, it is doubled and running on multiple cores before their successors, so all successor tasks get the results from their predecessors with lower communication costs. This type of algorithm has two disadvantages, a higher complexity (cubic, in relation to the number of tasks) and the duplication of the execution of tasks. Therefore they require more processor power [21, 23].
- In Guided-random schedulers, the scheduling influenced by policies applied to other sciences. For example, they are based on genetic algorithms or chemical reaction algorithms [23]. Their results can be improved if more repetitions are performed, which therefore makes it more expensive than the based Heuristic approach [21].

4.6 Basic Heterogeneous Scheduling Algorithms

4.6.1 HEFT

Topcuoglu et al [25] presents Heterogeneous Earliest-Finish-Time (HEFT) algorithm. It is one of the best, more acceptable and well documented list-based heuristic scheduling algorithm. HEFT is among the best schedule in compared between 20 scheduling heuristics algorithms as mention Canon et al [26]. At terms of complexity has a $O(n^2p)$ complexity, where n is the number of tasks and p the processors. The HEFT algorithm consists of two stages. First it assigns a priority to each task and all the tasks sorted in decreasing order. The tasks are ranked upwards and downwards to configure scheduling priorities. The maximum sum of computation and communication cost between the task and an exit node in the graph from that task is the upward rank of a task. The maximum sum of computation and communication cost between an entry node in the graph into that task is the downward rank. Afterwards, the task with the highest priority is assigned to the processor, which completes the execution of the task in the shortest possible time. Thus, the total execution time is minimized. There have been proposed various amendments of the above algorithm.

HEFT ALGORITMH
1: Compute $rank_u$ for all nodes
2: *ReadyTask* ← {Entry tasks}
3: While *ReadyTask* is not empty
4: Select the task *n* with highest priority
5: Assign the task *n* to the processor *p* that minimizes the *earliest finish time (EFT)* value of *n*
6: Update *earliest start time (EST)* values and *ReadyTask*
7: End while

4.6.2 CPOP

Critical Path on a Processor (CPOP) heuristic algorithm is proposed by Topcuoglu et al [25]. It also sorted the tasks in decreasing order. The tasks are ordered by the node priority arising to the addition of their upward rank and downward order. The task with the maximum sum belongs to the critical path. On each step these tasks are assigned to the critical-path-processor (CPP), which is usually the fastest processor that minimizes the length of the critical path, otherwise it is running on the processor that minimizes the EFT [23, 27]. The complexity is also as the HEFT, $O(n^2p)$.

```
CPOP ALGORITHM
1: Compute rank_u and rank_d for all nodes
2: ReadyTask ←{Entry tasks}
3: While ReadyTask is not empty
4:    Select the task n with highest priority
5:    If n is out the critical processor
6:          Assign n to the CPP
7:    Else
8:          Assign the task n to the processor P that
            minimizes the EFT value of n
9:    Update EST values and ReadyTask
10: End while
```

Both algorithms operate statically and require prior knowledge of computation and communication costs of each task for each processor type.

5. Implementation

The objective of this work is the implementation of a scheduling algorithm for dispatching jobs to accelerator processors in a heterogeneous system.

5.1 Basic Idea

The basic idea of the algorithm is as follows. In the context of a multiprogrammed environment we assume we have *N* applications, a set N_Q of queues with a fixed max weight that the system assigns to each queue. The weight of each queue represents the maximum time quantum allocated to this queue for execution, or the normalized maximum time quantum. Each application enqueues jobs in queues. The jobs are dispatched to a hardware accelerator either single or multi-threaded (e.g. GPU). Initially, we assume a single accelerator for servicing the works. The queue contains packets with the job attributes, such as pointers to the kernel code and data, estimated execution time, type, etc.

Special mention should be made to packet weight ($Weight_{(m)}$) and to the field *order*. The $Weight_{(m)}$ of each packet is different and independent from the weights of the other packets. The objective of *order* is to discern the enqueue order of a packet. Packets enqueued during the same cycle have the same *order.* If a packet has *order* value greater or equal than the *order* value of current cycle then the packet is ignored. This way the scheduler serves only the packets that have been enqueued when the cycle started. Packets enqueuing after the cycle is started will serve in the next cycle.

The centralized scheduler when deals with a queue with a higher weight (*weight_queue*) compared to a queue with a lower weight, must serve a number of packets in proportion to the ratio of their weights. Essentially, the algorithm scheduling policy works in Round-Robin fashion and provides weighted fairness for variable-length packets (i.e., execution time) maintained in multiple queues.

5.2 Algorithm Description

Each queue is designed as a ring buffer, with two pointers Front and Rear. In each queue fixed-size packets can be enqueued that contain the job weight (*Weight*) by amending the Rear pointer. A dequeue operation causes the Front pointer to decrease. If the Front pointer is equal to Rear then the queue is empty. If Front pointer plus one is equal to Rear pointer the queue is full. The above implementation is necessary due to the

limitation of physical memory. In our simulation model the queues are implemented as classical queues with head and tail pointers. In both cases the packets inside a queue follow a FIFO ordering.

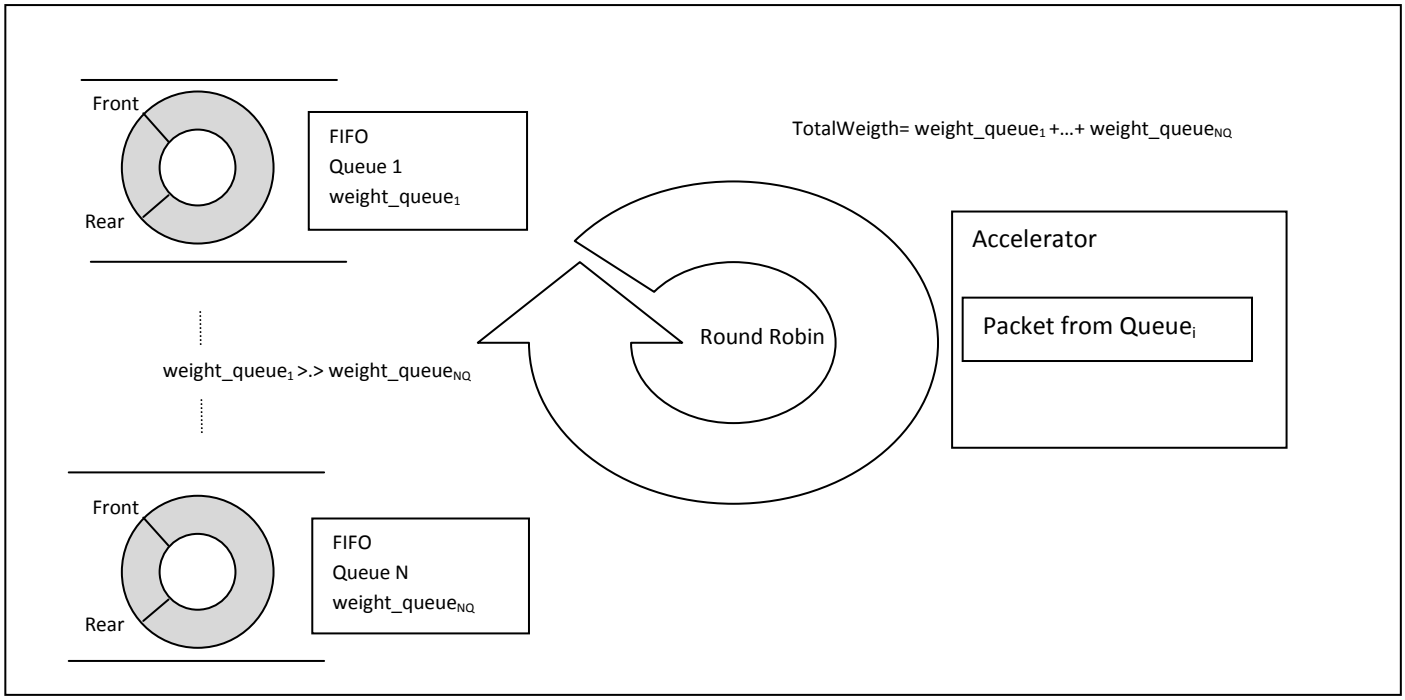

Figure 5.2-1 Schematic algorithm description

Every queue *i* ($1<=i<=N_Q$) has a maximum weight of dequeued packets ($Weight\_Queue_{(i)}$).The $Weight\_Queue_{(i)}$ is initially defined before the scheduler starts. Since, the sum of all weights ($Weight\_Queue_{(i)}$) determines the total weight (*TotalWeight).*

$$TotalWeight= Weight\_Queue_{(1)} + …+ Weight\_Queue_{(Q)} \quad (1)$$

Additionally there is a variable *current_order* which keeps the current order, which in turn is the same as the number of cycle. Another variable, the $order_{(packet\ m)}$, keeps the order when the packet is enqueued.

Each time it is possible to export packets as long as the following conditions are met with: a) the sum of total popped weight packets from all queues should not be exceeded by a maximum fixed threshold (*TotalWeight*) that was initially determined, b) the sum of the weights of the packets of each queue must be lower or equal than the weight of that queue ($Weight\_Queue_{(i)}$), and c) the $order_{(packet\ m)}$ must be less than current_order.

Also there is a variable or each queue *i* ($1<=i<=N_Q$) that keeps the current total weight ($CurrentTotalWeight_{(i)}$) of dequeued packets. $CurrentTotalWeight_{(i)}$ is initially set to zero. If a packet deqeue from a queue *i*, then the $CurrentTotalWeight_{(i)}$ is increased by the weight of packet.

$$CurrentTotalWeight_{(i)} = CurrentTotalWeight_{(i)} + Weight_{(m)} \quad (2)$$

Before being dequeued a packet, the sum of the current total weight of dequeued packets of the queue *i* plus the weight of the packet to be exported is calculated.

$$CurrentTotalWeight_{(i)} + Weight_{(m)} \quad (3)$$

If equation (3) is less than or equal to $Weight\_Queue_{(i)}$, the packet can be dequeued, while increasing the current total weight of dequeued packets by the weight of dequeued packet.

$$CurrentTotalWeight_{(i)} = CurrentTotalWeight_{(i)} + Weight_{(m)} \quad (4)$$

$$CurrentTotalWeight_{(i)} + Weight \quad (5)$$

If the equation (5) is more than $Weight\_Queue_{(i)}$ of queue *i*, then examine the next queue.

If the queue is empty then set the $CurrentTotalWeight_{(i)}$ equals $Weight\_Queue_{(i)}$ (equation 6)

$$CurrentTotalWeight_{(i)} = Weight\_Queue_{(i)} \quad (6)$$

Finally there is a global variable, *CurrentTotalWeight*, *which* increases by the $CurrentTotalWeight_{(i)}$ (equation 7)

$$CurrentTotalWeight = CurrentTotalWeight + CurrentTotalWeight_{(i)} \quad (7)$$

The necessary and sufficient conditions to dequeue one job are for the C*urrentTotalWeight* to be less than the *TotalWeight* and for the $CurrentTotalWeight_{(i)}$ to be less than the $Weight\_Queue_{(i)}$. Furthermore the $order_{(packet\ m)}$ must be less than the current order. The queue will become ineligible to serve a)if the *CurrentTotalWeight* is greater or equal to the *TotalWeight* or b) if the $CurrentTotalWeight_{(i)}$ is greater or equal to the $Weight\_Queue_{(i)}$ or c) the $order_{(packet\ m)}$ is greater than or equal to current order.

If the *CurrentTotalWeight* is greater or equal to the *TotalWeight*, it indicates that all packets permitted to be dequeued have indeed been dequeued. Also if the $CurrentTotalWeight_{(i)}$ is greater or equal to the $Weight\_Queue_{(i)}$ this denotes that all packets permitted to be dequeued from queue *i* have been actually dequeued.

In each cycle packets with maximum total weight equal to *TotalWeight* can be dequeued. If the *CurrentTotalWeight* value is the highest possible and simultaneously less than or equal to *TotalWeight* then a cycle is completed. After this cycle, the procedure starts over again by setting the *CurrentTotalWeight* and $CurrentTotalWeight_{(i)}$ for each queue *i* to zero. Thus, a new cycle begins for the packets that arrived during the previous cycle; the total system weight is still equal to *TotalWeight*. In this new cycle the scheduler examines the next eligible queue starting from the last queue plus one that was served in the previous cycle. This is done to provide fairness, since we avoid the bottom queues from starvation. Otherwise, with the start of every new cycle the first queue will be examined, consequently delaying the servicing of lower queues.

Algorithm WSOVL
Input: A set of N_Q Queues, $Weight\_Queue_{(i)}$, TotalWeight,
parameter queue number *i* to start from this queue.
Output: return a eligible queue number i,
or control value (-1 or -2)

Control Vlaue:
- -1: call again Algorithm… while not complete the requisite *total_weight*
- -2: empty queues or full weight

1:For each cycle do
2: If *Queues* are empty then return (-2);
3: While (*i*< N_Q) and (*packet not found)* do
4: If ((queue *i* is empty) or ($order_{(packet\ m)}$ >=*current order*)) and
($CurrentTotalWeight_{(i)}$< $Weight\_Queue_{(i)}$) then
5: update *CurrentTotalWeigth,* $CurrentTotalWeight_{(i)}$*;*
6: *e*xamine the next queue;
7: else if (queue *i* is not empty) and ($CurrentTotalWeight_{(i)}$ +

```
                                        Weight(packet)<=Weight_Queue(I))
            and (CurrentTotalWeight < TotalWeight) and (order(packet m)<current order)
      then
8:          found a packet in queue i;
9:          update CurrentTotalWeight(i); CurrentTotalWeight,
10:   else if (queue i is not empty) and (CurrentTotalWeight(i) +
                                        Weight(packet)>Weight_Queue(I))
            and (CurrentTotalWeight < TotalWeight) and (order(packet m) <current order)
      then
11:         update CurrentTotalWeight, CurrentTotalWeight(i);
12:         examine the next queue;
13:   else if (CurrentTotalWeight = TotalWeight) then
14:         return(-2);
15:   else
16:         examine the next queue;
17:   endif
18: End while.
19: If found a packet then
20:   If (i<N_Q) then
21:     will examine the next queue (q=i+1) on next cycle;
22:   else examine the queue 1 (q=1) on next cycle;
23:   endif
24:   return(i);
25: endif
26: if (i== N_Q) then return (-1);
27: End of Cycle;
```

6 Measurements

In order to study the behavior of the algorithm the following scenarios are implemented. These scenarios are indicative, with the purpose to demonstrate the behavior of algorithm. The algorithm can handle any number of queues but in the proof-concept examples, 8 queues have been assumed.

6.1 Scenario 1. As many packets as can be served in a cycle, average packet weight 100, minimum queue weight 300, maximum queue weight 1000.

The scenario is based on the following assumptions. Suppose we have 8 queues. Queue 0 (Q0) has the largest weight and queue 7 (Q7) the minimum weight. The weights of the queues have been assigned as follows: Q0:1000, Q1:900, Q2:800, Q3:700, Q4:600, Q5:500, Q6:400 and Q7:300. This means that Q0 can serve packets with total weight at most 1000, Q1 at most 900, and so on. The maximum total weight that can service in a cycle is the sum of the weights of all queues, which is 5200.

The weights of the packets that are enqeued into queues are provided by the Poisson distribution. The weight represents the execute time quanta of each packet. In our example the average weights of packets are 100. In each queue are enqeued as many packets as can be served in a cycle. For example in Q0, packets with maximum total weight equal to 1000 are enqeued, in Q1 packets with maximum total weight equal to 900 are enqeued, and so on. So Q0 has weight 1000 and can accept maximum 10 jobs of 100 each. The packets are enqeued just before the new cycle starts. This means that all queues are empty when the new packets are inserted. Moreover, we assume in all scenarios an additional delay of 2 time quanta of each job, because of reading, writing and transfer delay.

Table 6.1-1 shows the Mean Latency, Mean Weight, Max Latency, and Number of serviced Jobs, for 10026 total serviced packets.

Queue	Mean Latency	Mean Weight	Max Latency	Number of Jobs
0	3259.02238046796	100.021363173957	5032	1966
1	2645.64325842697	99.5921348314607	4926	1780
2	2485.47351627313	99.6119974473516	4723	1567
3	2293.92398523985	99.9372693726937	4437	1355
4	2074.53403141361	99.8368237347295	4095	1146

5	1815.50858369099	99.7907725321888	3607	932
6	1570.29769959405	99.2936400541272	3041	739
7	1255.66279069767	99.1027131782946	2353	516

Table 6.1-1 Results for as many packets as can be served in a cycle, average packet weight 100, min queue weight 300max queue weight 1000.

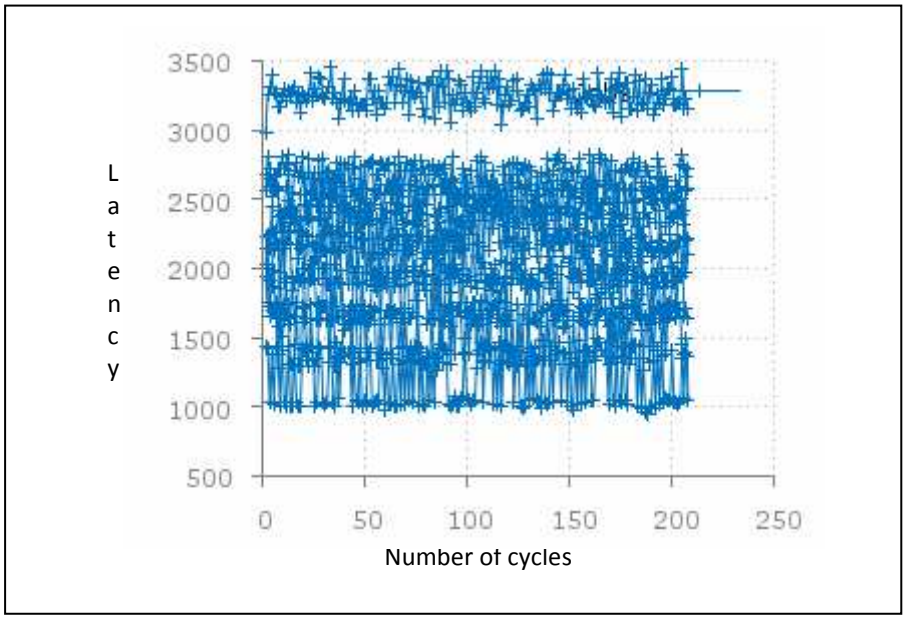

Figure 6.1-1 Latency for as many packets as can be served in a cycle average packet weight 100, min queue weight 300, max queue weight 1000.

From the above table we can understand that in such case, the queue with minimum weight (Q7. 300) enjoys the minimum latency. But Q7 serviced the fewest jobs. So if we desired to serve a small number of jobs with the minimum latency we would choose Q7.

If we have many jobs to service, then choose Q0. QO services the maximum jobs but simultaneously it has the maximum Mean Latency. Otherwise if we choose Q7 then the jobs will have to wait for more service cycles to complete.

In total we have 208 service cycles. In theses cycles we submit almost 9.45 jobs per service cycle to Q0, and 2.48 jobs per cycle to Q7 and 10026 total serviced jobs are submitted

6.2 Scenario 2. As many packets as can be served in a cycle, average packet weight 50, minimum queue weight 300, maximum queue weight 1000.

The weight of each queue, the total weight, the mode and the time of enqueing packets are the same as previous. The only difference is the weight of each packet. The weight is given by Poisson distribution; with distributed value 50, rather 100 in the previous scenario. So Q0 has weight 1000 and can accept maximum 20 jobs of 50 each, rather 10 jobs in scenario1, and so on.

Table 6.2-1 shows the Mean Latency, Mean Weight, Max Latency, and Number of serviced Jobs, for 10001 total serviced packets.

Queue	Mean Latency	Mean Weight	Max Latency	Number of Jobs
0	3397.7465648855	49.5648854961832	5263	1965
1	2954.39132915003	49.9566457501426	5131	1753
2	2750.22422680412	50.1024484536082	4949	1552
3	2523.56995581738	49.8063328424153	4736	1358
4	2223.09745390694	50.233538191396	4341	1139
5	1950.68376963351	49.9151832460733	3900	955
6	1628.28853754941	49.3333333333333	3404	759
7	1248.73211009174	50.0605504587156	2510	545

Table 6.2-1 Results for as many packets as can be served in a cycle, average packet weight 50, min queue weight 300, max queue weight 1000.

As in scenario 1, the queue with minimum weight (Q7. 300) enjoys the minimum latency. But Q7 serviced the fewest jobs. In addition QO services the maximum jobs but simultaneously it has the maximum Mean Latency.

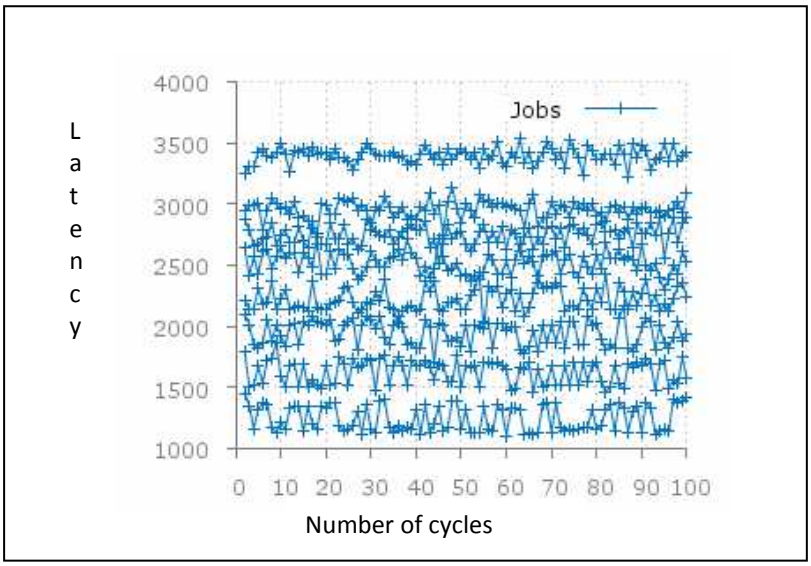

Figure 6.2-1 Latency for as many packets as can be served in a cycle average packet weight 50, min queue weight 300, max queue weight 1000.

The main anticipated difference between two scenarios is the service cycles. In scenario 1, 208 cycles are needed to service 10026 jobs. In scenario 2, 100 cycles are needed to service 10001 jobs. Hence, in about 200 cycles (as scenario 1) approximately 20000 jobs will be served, since we have smaller jobs.

6.3 Scenario 3. As many packets as can be served in a cycle, average packet weight 100, minimum queue weight 650, maximum queue weight 1000.

The weights of the queues have been assigned as follows: Q0:1000, Q1:950, Q2:900, Q3:850, Q4:800, Q5:750, Q6:700 and Q7:650. This means that Q0 can serve packets with total weight at most 1000, Q1 at most 950, and so on. The maximum total weight that can be serviced in a cycle is the sum of the weights of all queues, which is 6600. This implies that more jobs can served in a cycle, compared with the previous scenarios where the total weight was 5200.

The weights of the packets that are enqeued into queues are provided by the Poisson distribution. The weight represents the execute time quanta of each packet. In our example the average weights of packets are 100. In each queue are enqeued as many packets as can be served in a cycle. For example in Q0, packets with maximum total weight equal to 1000 are enqeued, in Q1 packets with maximum total weight equal to 950 are enqeued, and so on. So Q0 has weight 1000 and can accept maximum 10 jobs of 100 each. The packets are enqeued just before the new cycle starts. This means that all queues are

empty when the new packets are inserted. Moreover, we assume an additional delay of 2 times quanta of each job, because of reading, writing and transfer delay.

Table 6.3-1 shows the Mean Latency, Mean Weight, Max Latency, and Number of serviced Jobs, for 10031 total serviced packets.

Queue	Mean Latency	Mean Weight	Max Latency	Number of Jobs
0	3908.52272727273	99.8837662337662	6423	1540
1	3228.9780971937	99.8220396988364	6359	1461
2	3125.08127721335	99.7902757619739	6266	1378
3	3043.6499614495	99.7918272937548	6180	1297
4	2917.42737896494	100.510016694491	5981	1198
5	2863.1073943662	100.348591549296	5971	1136
6	2762.92768791627	99.7887725975262	5499	1051
7	2650.92680412371	99.9536082474227	5487	970

Table 6.3-1 Results for as many packets as can be served in a cycle, average packet weight 100, min queue weight 650, max queue weight 1000.

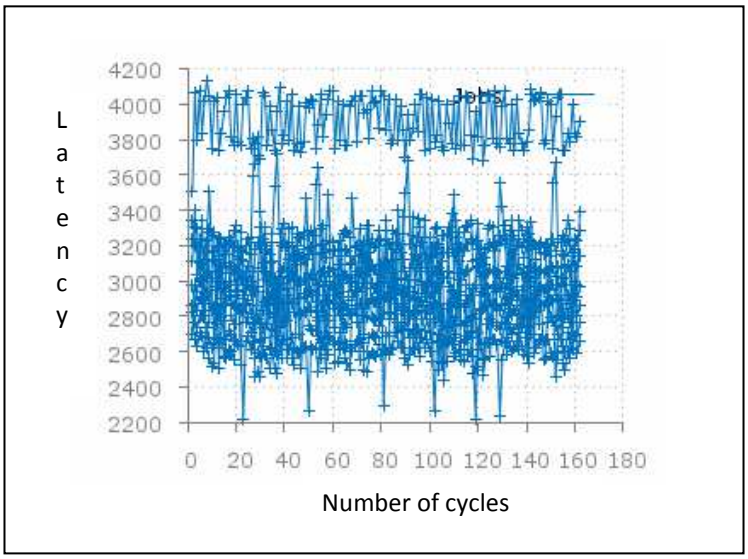

Figure 6.3-1 Latency for as many packets as can be served in a cycle average packet weight 100, min queue weight 650 max queue weight 1000.

For 10031 packets to be served were needed 162 cycles while in scenario 1 208 cycles were needed to serve 10026 jobs. The above result was expected since in each cycle more jobs are served. Identical with scenario 1, the queue with minimum weight (Q7. 650) enjoys the minimum latency. But Q7 serviced the fewest jobs. So if we desired to serve a small number of jobs with the minimum latency we would choose Q7. QO services the

maximum jobs but simultaneously it has the maximum Mean Latency. Otherwise if we choose Q7 then the jobs will have to wait for more service cycles to complete. Furthermore all queues served more jobs compared with scenario 1. Therefore by increasing the weight of a queue the number of the packets that can be served is increased.

6.4 Scenario 4. As many packets as can be served in a cycle, average packet weight 50, minimum queue weight 650, maximum queue weight 1000.

The weight of each queue, the total weight, the mode and the time of enqueing packets are the same as scenario 3. The only difference is the weight of each packet. The weight is given by Poisson distribution; with distributed value 50, rather 100 in previous scenario. So Q0 has weight 1000 and can accept maximum 20 jobs of 50 each, rather 10 jobs in scenario1, and so on.

Table 6.4-1 shows the Mean Latency, Mean Weight, Max Latency, and Number of serviced Jobs, for 10123 total serviced packets.

Queue	Mean Latency	Mean Weight	Max Latency	Number of Jobs
0	3970.19078520441	49.9539260220636	6700	1541
1	3564.08567511995	50.0575736806032	6672	1459
2	3422.30930064888	49.8500360490267	6633	1387
3	3315.99696279423	49.5512528473804	6515	1317
4	3152.15905383361	49.9363784665579	6353	1226
5	2969.58657243816	50.3533568904594	6056	1132
6	2864.81605975724	49.4743230625584	5970	1071
7	2704.33232323232	49.8464646464646	5555	990

Table 6.4-1 Results for as many packets as can be served in a cycle, average packet weight 50, min queue weight 650, max queue weight 1000.

.

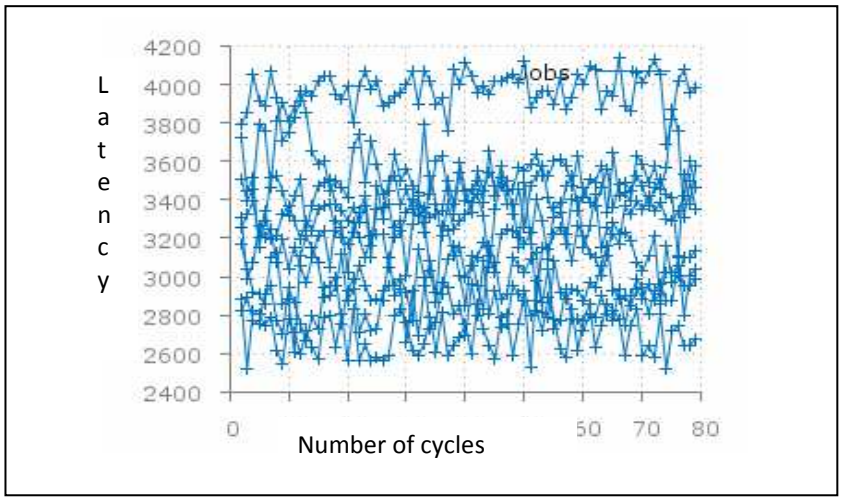

Figure 6.4-1 Latency for as many packets as can be served in a cycle average packet weight 50, min queue weight 650, max queue weight 1000.

As previous, the queue with minimum weight (Q7. 650) enjoys the minimum latency. But Q7 serviced the fewest jobs. In addition QO services the maximum jobs but simultaneously it has the maximum Mean Latency.

The main anticipated difference between scenario 3 and scenario 4 is the service cycles. In scenario 3, 162 cycles are needed to service 10031 jobs. In scenario 4, 79 cycles are needed to service 10123 jobs. Hence, in about 160 cycles (as scenario 3) will be served approximately 20200 jobs, since we have smaller jobs.

6.5 Scenario 5. Average packet weight 100, weight 300 of each queue.

We have 8 queues. Queue 0 (Q0) has the largest weight and queue 7 (Q7) the minimum weight. The weights of the queues have been assigned as follows: Q0:1000, Q1:900, Q2:800, Q3:700, Q4:600, Q5:500, Q6:400 and Q7:300. This means that Q0 can serve packets with total weight at most 1000, Q1 at most 900, and so on. The maximum total weight that can service in a cycle is the sum of the weights of all queues, which is 5200.

The weights of the packets that are enqeued into queues are provided by the Poisson distribution. The weight represents the execute time quanta of each packet. In our example the average weights of packets are 100. In each queue are enqeued packets with total weight equal to the minimum weight of the queues. The minimum weight is the weight of queue 7, which is 300. So in Q0, packets with maximum total weight equal to 300 are enqeued, in Q1 packets with maximum total weight equal to 300 are enqeued, and so on, although Q0 weight is 1000. So in a cycle 10 packets of 100 can be served. Nevertheless

maximum 3 jobs of 100 each are inserted. The same occurs with other queues. The packets are enqeued just before the new cycle starts. This means that all queues are empty when the new packets are inserted. Moreover, we assume an additional delay of 2 time quanta of each job, because of reading, writing and transfer delay.

Table 6.5-1 shows the Mean Latency, Mean Weight, Max Latency, and Number of serviced Jobs, for 1000 total serviced jobs.

Queue	Mean Latency	Mean Weight	Max Latency	Number of Jobs
0	956.067226890756	99.9747899159664	2052	119
1	1087.35658914729	98.2325581395349	2160	129
2	1041.11904761905	98.4761904761905	2138	126
3	1043.69230769231	98.7846153846154	2177	130
4	939.895161290323	99.2661290322581	2173	124
5	935.869918699187	100.032520325203	2063	123
6	940.301587301587	97.8174603174603	2052	126
7	973.09756097561	98.4065040650407	2139	123

Table 6.5-1 Results for packets with weight 300 of each queue, average packet weight 100.

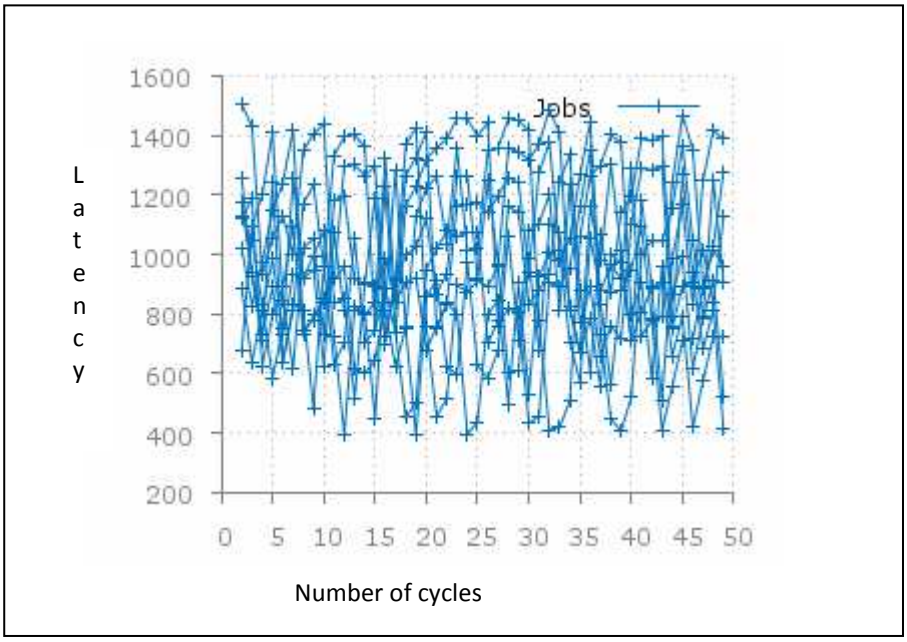

Figure 6.5-1 Latency for packets with weight 300 of each queue, average packet weight 100.

The number of enqueuing packets in each queue is approximately the same because there is no difference in the total weight of enqueuing packets in queues. The Mean

Latency, Max Latency, and Number of serviced Jobs are about the same. The Mean Latency range from 935,86 to 1087,35, Max Latency range from 2052 to 2160 and the Number of serviced Jobs range from 123 to 130. Moreover 49 cycles are needed to serve 1000 jobs. Accordingly 490 cycles are needed to serve 10000 jobs. Additionally the most cycles needed than all previous scenarios. Briefly in such circumstance the algorithm behaves as Round Robin algorithm.

6.6 Scenario 6. Average packet weight 100, weight 1000 of each queue.

We have 8 queues. Queue 0 (Q0) has the largest weight and queue 7 (Q7) the minimum weight. The weights of the queues have been assigned as follows: Q0:1000, Q1:900, Q2:800, Q3:700, Q4:600, Q5:500, Q6:400 and Q7:300. This means that Q0 can serve packets with total weight at most 1000, Q1 at most 900, and so on. The maximum total weight that can service in a cycle is the sum of the weights of all queues, which is 5200.

The weights of the packets that are enqeued into queues are provided by the Poisson distribution. The weight represents the execute time quanta of each packet. In our example the average weights of packets are 100. In each queue are enqeued packets with total weight equal to the maximum weight of the queues. The maximum weight is the weight of queue 0, which is 1000. So in Q0, packets with maximum total weight equal to 1000 are enqeued, in Q1 packets with maximum total weight equal to 1000 are enqeued, and so on, although Q7 weight is 300. So in a cycle 3 packets of 100 can be served. Nevertheless about 10 jobs of 100 each are inserted. The same occurs with queues 1-6. The packets are enqeued just before the new cycle starts. This means that all queues are empty when the new packets are inserted. Moreover, we assume an additional delay of 2 time quanta of each job, because of reading, writing and transfer delay.

Table 6.6-1 shows the Mean Latency, Mean Weight, Max Latency, and Number of serviced Jobs, for 1007 total serviced jobs.

Queue	Mean Latency	Mean Weight	Max Latency	Number of Jobs
0	3230.16243654822	100.345177664975	4946	197
1	8039.74725274725	99.4340659340659	14665	182
2	9138.80503144654	99.4276729559748	17215	159
3	9396.40875912409	100.401459854015	17309	137

4	10653.4086956522	100.191304347826	20613	115
5	12724.8888888889	102.088888888889	25448	90
6	12083.8	98.4933333333333	25550	75
7	18095.3076923077	99.2115384615385	30650	52

Table 6.6-1 Results for packets with weight 1000 of each queue, average packet weight 100.

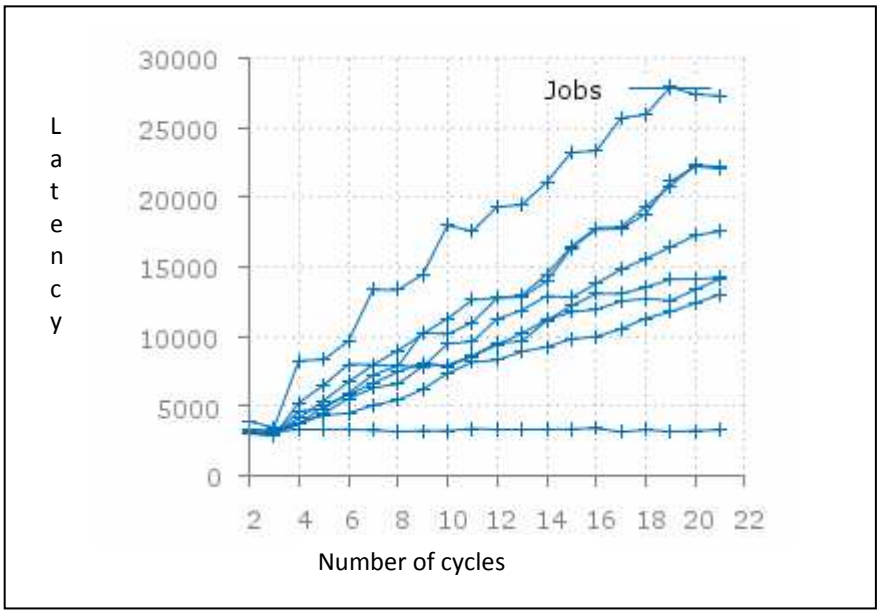

Figure 6.6-1 Latency for packets with weight 1000 of each queue, average packet weight 100.

QO has the minimum Mean Latency 3230.16 and minimum Max Latency 4946. The Mean Latency rate and Max Latency are dramatically increasing in other queues. Hence Q7 has the maximum Mean Latency 18095.30 and maximum Max Latency 30650. This is a consequence of the number of packets enqueuing in a cycle, as in a cycle enqueuing more packets than can be served in a cycle. So packets in Q1-Q7 must wait for the next cycles to serve. Therefore a bottleneck is happening in these queues. This effect was mostly pronounced in lightweight queues with Q7 that have the major problem. Also Q0 served more jobs (197) than other queues. Q7 served fewer jobs (52) than other queues. Also 21 cycles are needed to serve 1000 jobs. Accordingly 210 cycles are needed to serve 10000 jobs.

6.7 Scenario 7. Average packet weight 50 for Q0-Q3, average packet weight 100 for Q4-Q7, minimum queue weight 300, maximum queue weight 1000.

We have 8 queues as above. The weights of the queues have been assigned as follows: Q0:1000, Q1:900, Q2:800, Q3:700, Q4:600, Q5:500, Q6:400 and Q7:300. The weights of the packets that are enqeued into queues are provided by the Poisson distribution. The queues 0, 1, 2 and 3 the average weights of packets are 50. The queues 4, 5, 6 and 7 the average weights of packets are 100. So in Q0, above 20 packets with maximum total weight equal to 1000 are enqeued, in Q1 above 18 packets with maximum total weight equal to 900 are enqeued, and so on for queue 2 and 3. Q4 has weight 600, so no more than 6 packets with maximum total weight equal to 600 are enqeued, in Q5 above 5 packets with maximum total weight equal to 500 are enqeued, and so on for queue Q4 and Q7. In all cases the packets are enqeued just before the new cycle starts. This means that all queues are empty when the new packets are inserted. Moreover, we assume an additional delay of 2 time quanta of each job, because reading, writing and transfer delay.

Table 6.7-1 shows the Mean Latency, Mean Weight, Max Latency, and Number of serviced Jobs, for 10037 total serviced jobs, in 101 cycles.

Queue	Mean Latency	Mean Weight	Max Latency	Number of Jobs
0	4392.73860182371	49.9620060790274	6626	1974
1	4026.76063829787	49.738829787234	6539	1880
2	3911.38951521984	49.7519729425028	6498	1774
3	3818.30796884362	49.8933493109647	6352	1669
4	2156.60342555995	99.6758893280632	4421	759
5	2107.1946403385	100.056417489422	4412	709
6	2075.47734138973	99.404833836858	4119	662
7	2011.12786885246	99.6590163934426	4135	610

Table 6.7-1 Results for packets with minimum queue weight 300, maximum queue weight 1000, average packet weight 50 for Q0-Q3, average packet weight 100 for Q4-Q7.

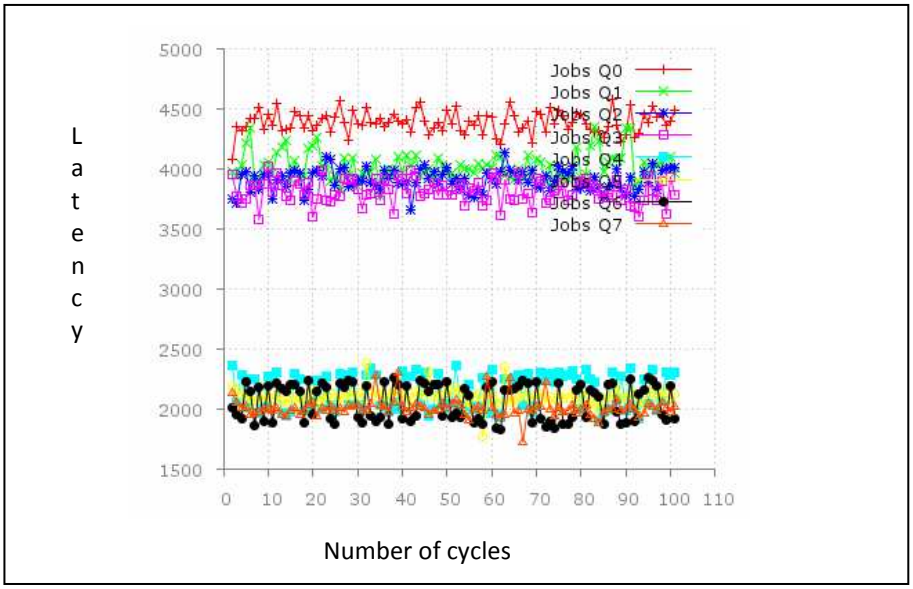

Figure 6.7-1 Latency for packets with minimum queue weight 300, maximum queue weight 1000, average packet weight 50 for Q0-Q3, average packet weight 100 for Q4-Q7.

The Mean Latency between Q0, Q1, Q2 and Q3 ranges between 3818,30 and 4392,73. Also Max Latency ranges between 6352 and 6626. Q0 has the maximum Mean Latency and Max Latency. Q3 has the minimum Mean Latency and minimum Max Latency. But Q0 has services most jobs, 1974, rather Q3 which has serviced 1669 jobs, of about 50 each.

For Q4, Q5, Q6 and Q7 the Mean Latency ranges between 2011,12 and 2156,60. Also Max Latency ranges between 4135 and 4421. Q4 has the maximum Mean Latency and maximum Max Latency. Q7 has the minimum Mean Latency and Q6 has the minimum Max Latency. But Q4 has services most jobs, 759, rather Q7 which has serviced 610 jobs, of about 100 each.

From all queues Q0 has the maximum Mean Latency and Max Latency. Q7 has the minimum Mean Latency and Max Latency. But Q0 services most jobs 1974, of 50 each, rather Q7 which services 610 jobs, of 100 each.

So if we have many small jobs we will choose Q0. If we have fewer small jobs and we care about latency we will choose Q3. If we have many large jobs we will choose Q4. If we have fewer large jobs and we care about latency we will choose Q7.

6.8 Scenario 8. Average packet weight 100 for Q0-Q3, average packet weight 50 for Q4-Q7, minimum queue weight 300, maximum queue weight 1000.

We have 8 queues as above. The weights of the queues have been assigned as follows: Q0:1000, Q1:900, Q2:800, Q3:700, Q4:600, Q5:500, Q6:400 and Q7:300. The weights of the packets that are enqeued into queues are provided by the Poisson distribution. For queues 0, 1, 2 and 3 the average weights of packets are 100. The queues 4, 5, 6 and 7 the average weights of packets are 50. So in Q0, above 10 packets with maximum total weight equal to 1000 are enqeued, in Q1 above 9 packets with maximum total weight equal to 900 are enqeued, and so on for queue 2 and 3. Q4 has weight 600, so about 12 packets with maximum total weight equal to 600 are enqeued, in Q5 above 10 packets with maximum total weight equal to 500 are enqeued, and so on for queues Q4 and Q7. In all cases the packets are enqeued just before the new cycle starts. This means that all queues are empty when the new packets are inserted. Moreover, we assume an additional delay of 2 time quanta of each job, because reading, writing and transfer delay.

Table 6.8-1 shows the Mean Latency, Mean Weight, Max Latency, and Number of serviced Jobs, for 10051 total serviced jobs, in 110 cycles.

Queue	Mean Latency	Mean Weight	Max Latency	Number of Jobs
0	2776.37523809524	99.7114285714286	5676	1050
1	2713.11212121212	100.083838383838	5564	990
2	2673.33724653148	99.6029882604056	5377	937
3	2629.11173814898	99.8092550790068	5393	886
4	4295.09613130129	49.8886283704572	6608	1706
5	3804.53875	49.86	6619	1600
6	3650.17222963952	49.6829105473965	6493	1498
7	3485.12066473988	49.7832369942197	6387	1384

Table 6.8-1 Results for packets with minimum queue weight 300, maximum queue weight 1000, average packet weight 100 for Q0-Q3, average packet weight 50 for Q4-Q7.

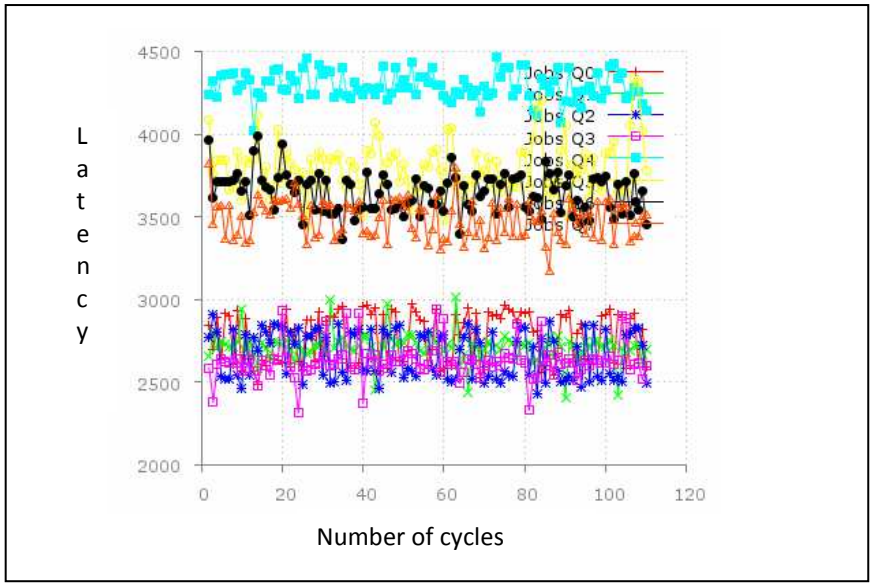

Figure 6.8-1 Latency for packets with minimum queue weight 300, maximum queue weight 1000, average packet weight 100 for Q0-Q3, average packet weight 50 for Q4-Q7.

The Mean Latency between Q0, Q1, Q2 and Q3 ranges between 2629,11 and 2776,35. Also Max Latency ranges between 5676 and 5377. Q0 has the maximum Mean Latency and Max Latency. Q3 has the minimum Mean Latency and Q2 has the minimum Max Latency. But Q0 has serviced most jobs, 1050, rather than Q3 which has serviced 886 jobs, of about 100 each.

For Q4, Q5, Q6 and Q7 the Mean Latency ranges between 3485,12 and 4295,09. Also Max Latency ranges between 6387 and 6619. Q4 has the maximum Mean Latency. Q5 maximum Max Latency, Q4 is next with very small difference (6608). Q7 has the minimum Mean Latency and the minimum Max Latency. But Q4 has serviced most jobs, 1706, rather Q7 which has serviced 1384 jobs, of about 50 each.

So if we have many small jobs we will choose Q4. If we have fewer small jobs and we care about latency we will choose Q7. If we have many large jobs we will choose Q1. If we have fewer large jobs and we care about latency we will choose Q3.

6.9 Scenario 9. Average packet weight 100, insert 3 new packets every 800 quanta, minimum queue weight 300, maximum queue weight 1000.

The scenario is based on the following assumptions. Suppose we have 8 queues. Queue 0 (Q0) has the largest weight and queue 7 (Q7) the minimum weight. The weights of the queues have been assigned as follows: Q0:1000, Q1:900, Q2:800, Q3:700, Q4:600, Q5:500, Q6:400 and Q7:300. This means that Q0 can serve packets with total weight at most

1000, Q1 at most 900, and so on. The maximum total weight that can service in a cycle is the sum of the weights of all queues, which is 5200.

The weights of the packets that are enqeued into queues are provided by the Poisson distribution. The weight represents the execute time quanta of each packet. In our example the average weights of packets are 100. The new packets are enqueued every 800 quanta. The total number of new packets is 3 for each queue. This means that the packets are enqeued during the cycle and that queues are non empty when the new packets are inserted. So the new packets will be served in the next cycle. Additionally the enqueuing of packets take place about 6,5 times per cycle. This means that in each cycle totally 19 packets, with 1900 total weights are enqueued in each queue. Hence in each queue equeue more packets than can be serviced are enqueued.

Moreover, we assume an additional delay of 2 time quanta of each job, because of reading, writing and transfer delay.

Table 6.9-1 shows the Mean Latency, Mean Weight, Max Latency, and Number of serviced Jobs, for 10016 total serviced packets.

Queue	Mean Latency	Mean Weight	Max Latency	Number of Jobs
0	93397.5945945946	99.2607607607608	179038	1998
1	139561.345546787	100.033258173619	272308	1774
2	180265.377394636	99.8690932311622	359649	1566
3	226233.677753141	100.311899482631	449688	1353
4	270598.750437828	100.039404553415	537827	1142
5	309950.241596639	99.3518907563025	617593	952
6	354316.690277778	100.319444444444	714030	720
7	400926.692759296	100.195694716243	800971	511

Table 6.9-1 Average packet weight 100, insert 3 new packets every 800 quanta, minimum queue weight 300, maximum queue weight 1000.

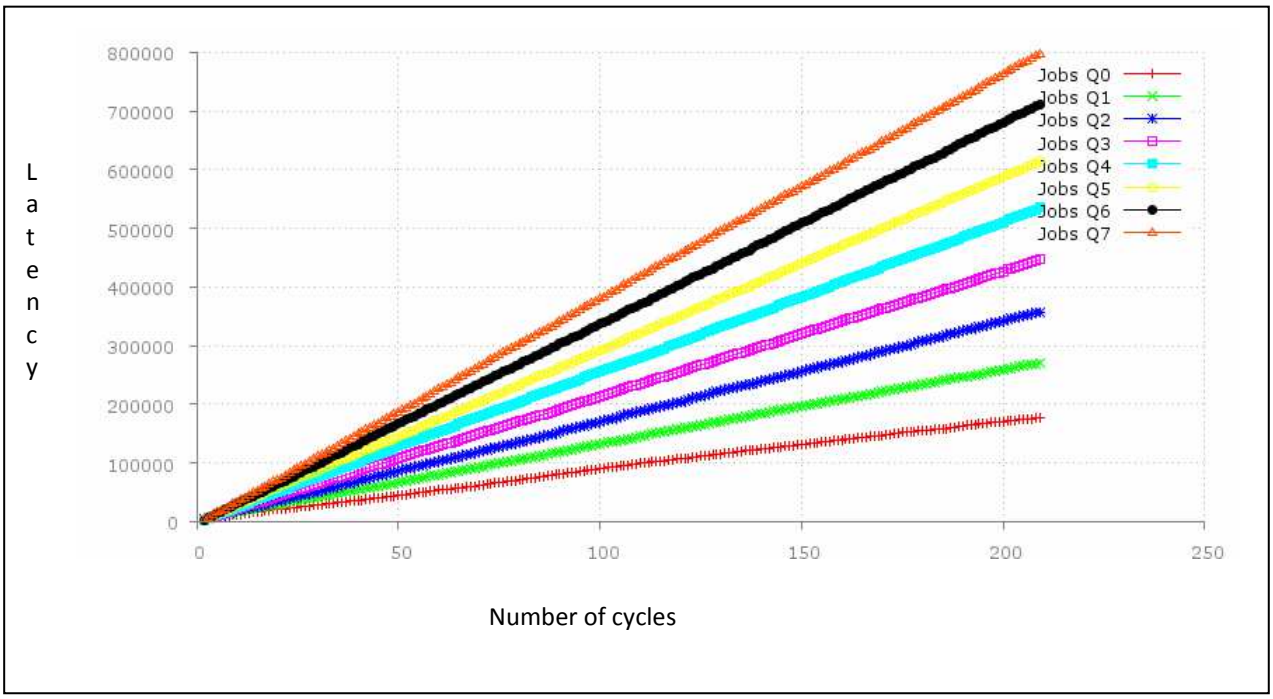

Figure 6.9-1 Latency for average packet weight 100, insert 3 new packets every 800 quanta, minimum queue weight 300, maximum queue weight 1000.

From the above table we can understand that in such case, the queue with maximum weight (Q0, 1000) enjoys the minimum latency and the maximum number of serviced jobs. Q7 serviced the fewest jobs with the maximum latency. So if we desired many jobs to service with the minimum latency we would choose Q0. Otherwise if we choose Q7 then fewer jobs will have to wait for more service cycles to complete. In total we have 209 service cycles and 10016 total serviced jobs are submitted.

6.10 Scenario 10. Average packet weight 100, insert 3 new packets every 800 quanta, minimum queue weight 650, maximum queue weight 1000.

We have 8 queues. Queue 0 (Q0) has the largest weight and queue 7 (Q7) the minimum weight. The weights of the queues have been assigned as follows: Q0:1000, Q1:950, Q2:900, Q3:850, Q4:800, Q5:750, Q6:700 and Q7:650. This means that Q0 can serve packets with total weight at most 1000, Q1 at most 950, and so on. The maximum total weight that can service in a cycle is the sum of the weights of all queues, which is 6600.

The weights of the packets that are enqeued into queues are provided by the Poisson distribution. The weight represents the execute time quanta of each packet. In our

example the average weights of packets are 100. So the new packets will be served in the next cycle.

3 new packets are enqueued every 800 quanta. This means that the packets are enqeued during the cycle and that queues are non empty when the new packets are inserted. Additionally the enqueuing of packets take place about 8 times per cycle. This means that in each cycle totally 24 packets, with 2400 total weights, are enqueued in each queue. Hence in each queue are equeued more packets than can be serviced.

Moreover, we assume an additional delay of 2 time quanta of each job, because of reading, writing and transfer delay.

Table 6.10-1 shows the Mean Latency, Mean Weight, Max Latency, and Number of serviced Jobs, for 10035 total serviced packets and 162 service cycles

Queue	Mean Latency	Mean Weight	Max Latency	Number of Jobs
0	190076.655642023	99.6634241245136	375360	1542
1	207838.589690722	100.039175257732	411114	1455
2	223753.073454545	99.8138181818182	444903	1375
3	239614.865637066	100.350579150579	477960	1295
4	258385.193574959	100.000823723229	511825	1214
5	274210.406660824	99.4609991235758	542454	1141
6	292585.232535885	100.444019138756	582382	1045
7	309263.135330578	99.7944214876033	614371	968

Table 6.10-1 Average packet weight 100, insert 3 new packets every 800 quanta, minimum queue weight 650, maximum queue weight 1000.

As in scenario 9 the queue with maximum weight (Q0, 1000) enjoys the minimum latency and the maximum number of serviced jobs. Q7 serviced the fewest jobs with the maximum latency. So if we desired many jobs to service with the minimum latency we would choose Q0. Otherwise if we choose Q7 then fewer jobs will have to wait for more time to be completed. In total we have 209 service cycles and 10034 total serviced jobs are submitted.

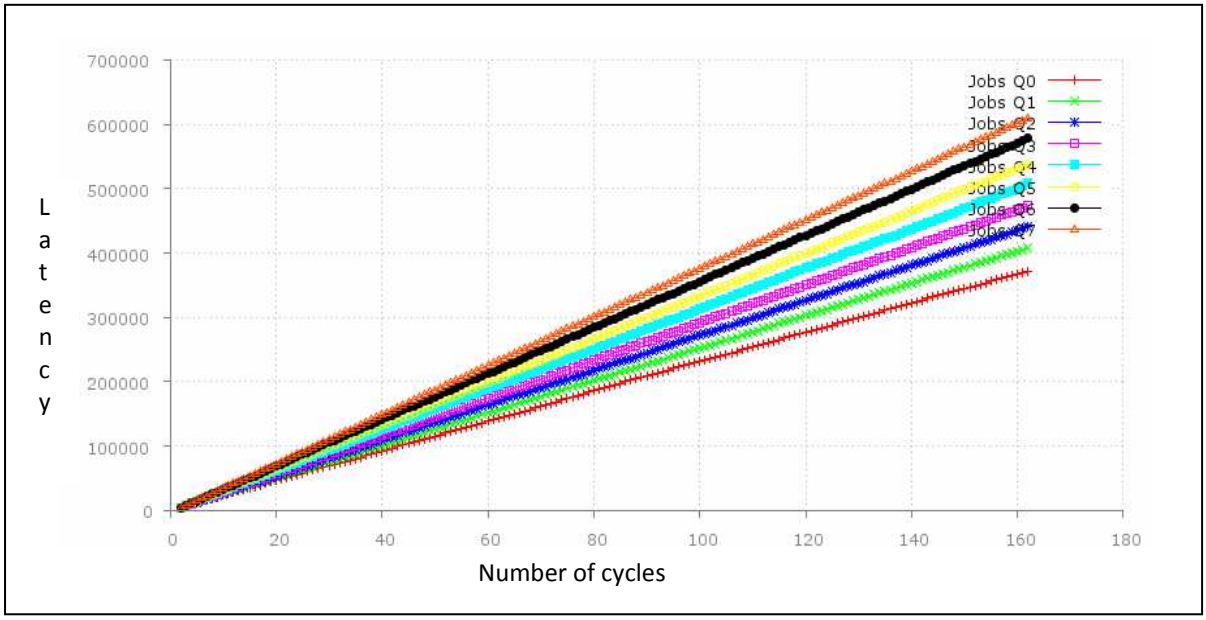

Figure 6.10-1 Latency for average packet weight 100, insert 3 new packets every 800 quanta, minimum queue weight 650, maximum queue weight 1000.

In comparison with scenario 9 (where about 10000 packets were served), Q0-Q3 have greater latency and fewer jobs served. On the other hand, Q3-Q7 have less latency but more jobs served. Although the serviced packets are almost the same the number of cycle is less (162 instead of 209). If we assume the same number of cycle then more jobs per queue will be served, in comparison to scenario 9. This is due to queues weight, which is greater in scenario 10, than in scenario 9.

6.11 Scenario 11. Average packet weight 50, insert 3 new packets every 800 quanta, minimum queue weight 650, maximum queue weight 1000.

We have 8 queues. Queue 0 (Q0) has the largest weight and queue 7 (Q7) the minimum weight. The weights of the queues have been assigned as follows: Q0:1000, Q1:950, Q2:900, Q3:850, Q4:800, Q5:750, Q6:700 and Q7:650. This means that Q0 can serve packets with total weight at most 1000, Q1 at most 950, and so on. The maximum total weight that can service in a cycle is the sum of the weights of all queues, which is 6600.

The weights of the packets that are enqeued into queues are provided by the Poisson distribution. The weight represents the execute time quanta of each packet. In our example the average weights of packets are 50. So the new packets will be served in the next cycle.

3 new packets are enqueued every 800 quanta. This means that the packets are enqeued during the cycle and that queues are non empty when the new packets are inserted. Additionally the enqueuing of packets take place about 8 times per cycle. This means that in each cycle totally 24 packets, with 2400 total weights, are enqueued in each queue. Hence in each queue are equeued more packets than can be serviced.

Moreover, we assume an additional delay of 2 time quanta of each job, because of reading, writing and transfer delay.

Table 6.11-1 shows the Mean Latency, Mean Weight, Max Latency, and Number of serviced Jobs, for 10075 total serviced packets.

Queue	Mean Latency	Mean Weight	Max Latency	Number of Jobs
0	5280.03301886792	49.2767295597484	6781	1272
1	5315.37716535433	50.3748031496063	6491	1270
2	5368.7474429583	50.1054287962234	6633	1271
3	5418.1332807571	49.98738170347	6493	1268
4	5470.12066246057	50.057570977918	6549	1268
5	5544.24861878453	49.8468823993686	6999	1267
6	6534.18304278922	50.4112519809826	8802	1262
7	20283.7593984962	49.7852965747703	33894	1197

Table 6.11-1 Average packet weight 50, insert 3 new packets every 800 quanta, minimum queue weight 650, maximum queue weight 1000.

From the above table we can work out that in such case, the queue with maximum weight (Q0, 1000) enjoys the minimum latency and the maximum number of serviced jobs, but with little difference compared with queues Q1-Q5. Q7 serviced the fewest jobs, as expected, with the maximum latency, about five times greater than Q0. Q6 achieves better performance in relation to Q7. So if we desired many jobs to be serviced with the minimum latency we would choose Q0 and then one queue of Q1-Q5. Otherwise if we choose Q7, then fewer jobs will have to wait for more time to complete. Also Q7 achieves the worst performance due to large number of enqeuing packets (24 per cycle), while just 13 packets may be serviced per cycle. In total we have 95 service cycles and 10075 total serviced jobs are submitted.

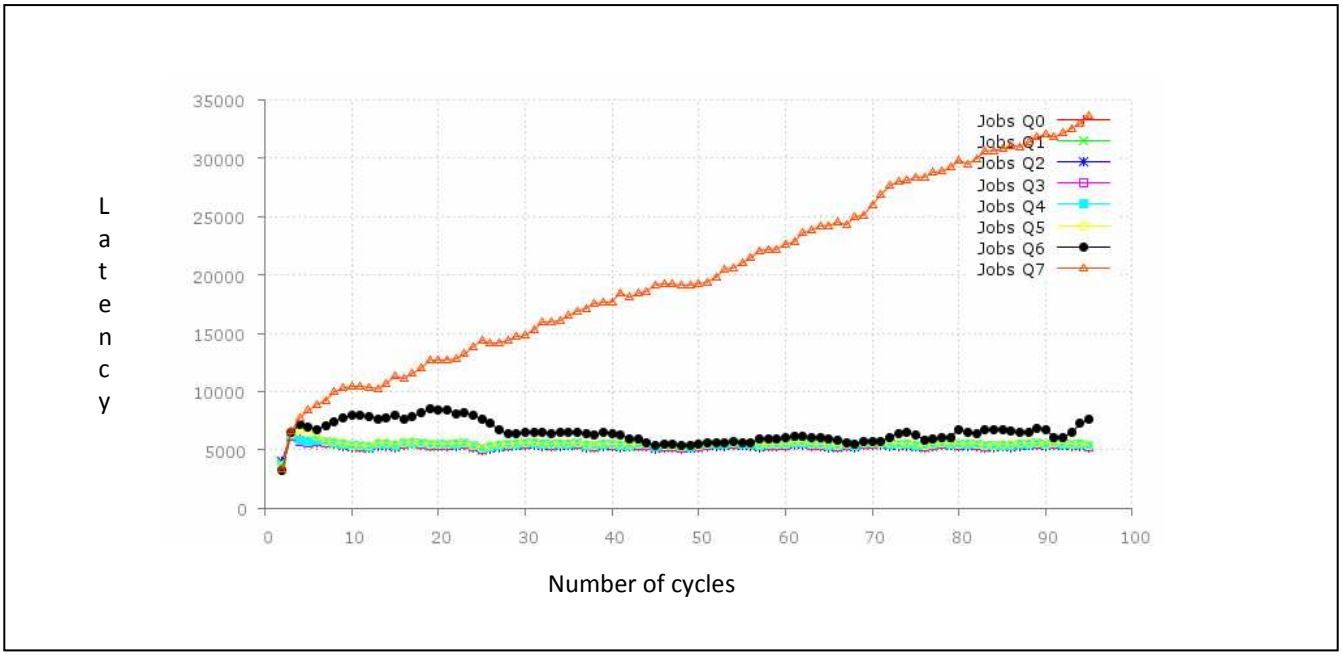

Figure 6.11-1 Latency for average packet weight 50, insert 3 new packets every 800 quanta, minimum queue weight 650, maximum queue weight 1000.

6.12 Scenario 12. Average packet weight 50, insert 3 new packets every 800 quanta, minimum queue weight 300, maximum queue weight 1000.

As in the previous scenario, but in this example the weights of the queues have been assigned as follows: Q0:1000, Q1:900, Q2:800, Q3:700, Q4:600, Q5:500, Q6:400 and Q7:300. The average weights of packets are 50. Table 6.12-1 shows the Mean Latency, Mean Weight, Max Latency, and Number of serviced Jobs, for 10036 total serviced packets, in 208 cycles.

Queue	Mean Latency	Mean Weight	Max Latency	Number of Jobs
0	2252.02821316614	49.2766457680251	5232	1276
1	2284.76983503535	50.3794186959937	5066	1273
2	2334.9677672956	50.1147798742138	4949	1272
3	2383.08818897638	49.9370078740157	4981	1270
4	2430.23599052881	50.0568271507498	5034	1267
5	2516.33412322275	49.8515007898894	5276	1266
6	3039.63370253165	50.3773734177215	9051	1264
7	34546.206445993	49.8214285714286	48875	1148

Table 6.12-1 1 Average packet weight 50, insert 3 new packets every 800 quanta, minimum queue weight 300, maximum queue weight 1000.

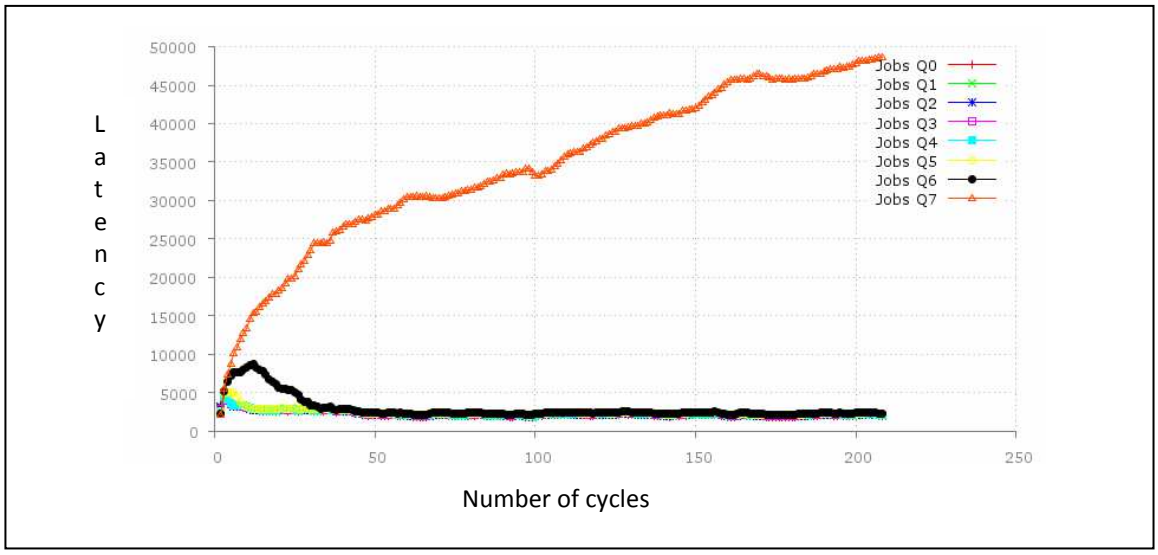

Figure 6.12-1 1 Latency for average packet weight 50, insert 3 new packets every 800 quanta, minimum queue weight 300, maximum queue weight 1000.

In comparison with scenario 11 208 cycles were needed for 10036 packets to be served rather 95 cycles for 10075 packets to be served. This seems reasonable since the weights of packets are less so more cycles are needed to serve the same number of jobs. The queue with maximum weight (Q0, 1000) enjoys the maximum number of serviced jobs and the minimum latency, but with little difference compared with queues Q1-Q5. Q7 serviced the fewest jobs, as expected, with the maximum latency, about five times greater than Q0. This is due to the large number of enqeuing packets (24 per cycle) while only 13 packets may be serviced per cycle. Q6 achieves better performance in relation to Q7. In total we have 208 service cycles and 10036 total serviced jobs are submitted.

The following measurements are based on the same scenario. At the beginning all the queues are full with the maximum number of packets that can be served in a cycle. Each time a packet is dequeued then a new packet is enqueued in the same queue. In this way the queues are always full. The new packets will be served in the next cycle.

6.13 Scenario 13. Average packet weight 100, minimum queue weight 300, maximum queue weight 1000. Each time a packet is dequeued then a new packet is enqueued in the same queue.

Suppose we have 8 queues. Queue 0 (Q0) has the largest weight and queue 7 (Q7) the minimum weight. The weights of the queues have been assigned as follows: Q0:1000, Q1:900, Q2:800, Q3:700, Q4:600, Q5:500, Q6:400 and Q7:300. This means that Q0 can serve

packets with total weight at most 1000, Q1 at most 900, and so on. The maximum total weight that can service in a cycle is the sum of the weights of all queues, which is 5200.

The weights of the packets that are enqeued into queues are provided by the Poisson distribution. The weight represents the execute time quanta of each packet. In our example the average weights of packets are 100. Table 6.13-1 shows the Mean Latency, Mean Weight, Max Latency, and Number of serviced Jobs, for 10003 total serviced packets, in 225 cycles.

Queue	Mean Latency	Mean Weight	Max Latency	Number of Jobs
0	4426.63968795709	98.2662116040956	5572	2051
1	4423.33920704846	98.454295154185	4768	1816
2	4422.75031446541	98.8509433962264	4778	1590
3	4421.88619676946	98.298825256975	4768	1362
4	4420.99383259912	98.2105726872247	4779	1135
5	4419.76670317634	98.7338444687842	4755	913
6	4416.56304985337	99.0630498533724	4751	682
7	4417.45374449339	99.2599118942731	4755	454

Table 6.13-1 Each time a packet is dequeued then a new packet is enqueued in the same queue. Average packet weight 100, minimum queue weight 300, maximum queue weight 1000.

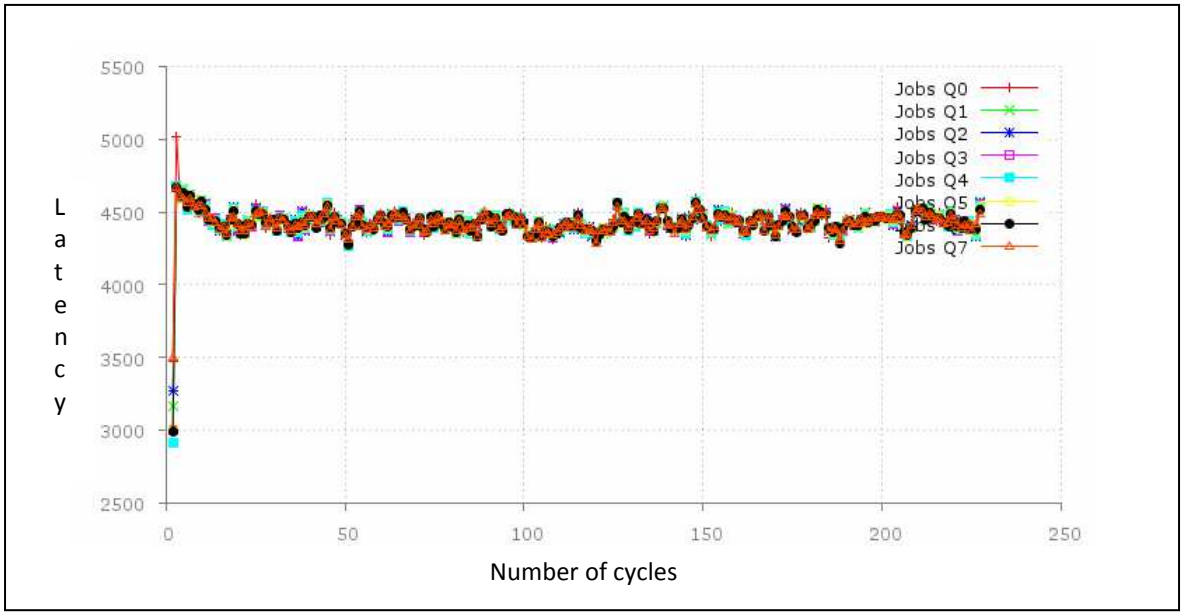

Figure 6.13-1 Latency for each time a packet is dequeued then a new packet is enqueued in the same queue. Average packet weight 100, minimum queue weight 300, maximum queue weight 1000.

The Mean Latency is approximately the same for all queues. Q0 has the maximum latency. This occurs in the first cycle so it isn't representative. Q0 has serviced most jobs, 2051, next is Q1 which has serviced 1816 jobs, and so on. Q7 has serviced fewer jobs, 454. So Q0 has the best performance followed by the other queues.

6.14 Scenario 14. Average packet weight 50, minimum queue weight 300, maximum queue weight 1000. Each time a packet is dequeued then a new packet is enqueued in the same queue.

As in the previous scenario, but in this example the average weights of packets are 50. Table 6.14-1 shows the Mean Latency, Mean Weight, Max Latency, and Number of serviced Jobs, for 10065 total serviced packets, in 106 cycles.

Queue	Mean Latency	Mean Weight	Max Latency	Number of Jobs
0	4837.83384615385	48.9389743589744	5615	1950
1	4832.87843137255	48.5058823529412	5233	1785
2	4831.48795944233	48.8384030418251	5241	1578
3	4829.6754194019	48.4106491611962	5230	1371
4	4826.44069264069	48.0337662337662	5234	1155
5	4824.74661105318	48.8571428571429	5227	959
6	4818.84124830393	48.4803256445048	5217	737
7	4815.26603773585	48.3169811320755	5207	530

Table 6.14-1 Each time a packet is dequeued then a new packet is enqueued in the same queue. Average packet weight 50, minimum queue weight 300, maximum queue weight 1000.

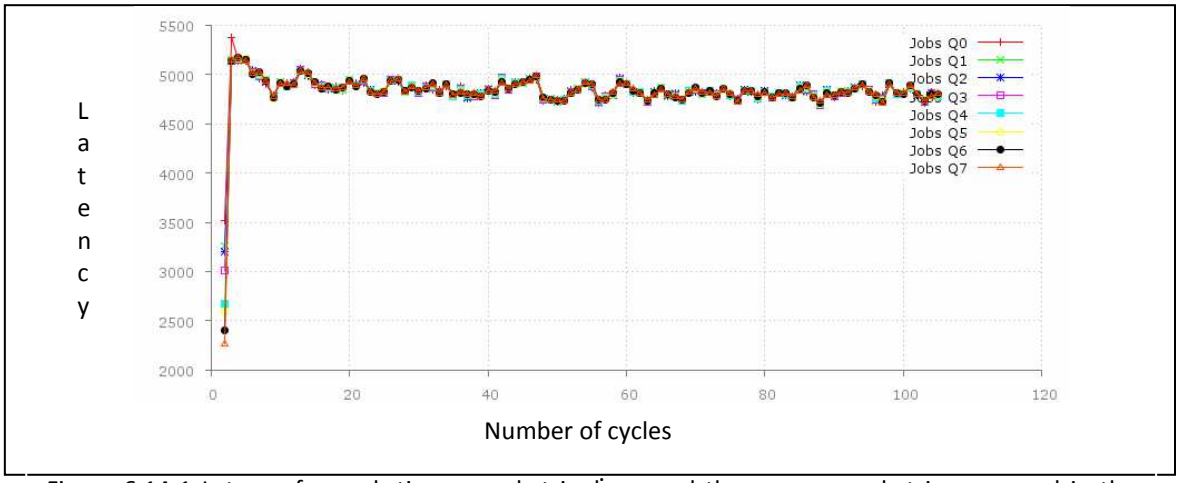

Figure 6.14-1 Latency for each time a packet is dequeued then a new packet is enqueued in the same queue. Average packet weight 50, minimum queue weight 300, maximum queue weight 1000

The conclusions from Table 6.14-1 are the same as in scenario 13. Q0 has serviced most jobs, 1950, next is Q1 which has serviced 1785 jobs, and so on. Q7 has serviced fewer jobs, 530. So Q0 has the best performance followed by the other queues.

6.15 Scenario 15. Average packet weight 100, minimum queue weight 650, maximum queue weight 1000. Each time a packet is dequeued then a new packet is enqueued in the same queue.

The weights of the queues have been assigned as follows: Q0:1000, Q1:950, Q2:900, Q3:850, Q4:800, Q5:750, Q6:700 and Q7:650. The maximum total weight that can be serviced in a cycle is the sum of the weights of all queues, which is 6600. This implies that more jobs can be served in a cycle, compared to the previous scenarios where the total weight was 5200. In our example, the average weights of packets are 100. Table 6.15-1 shows the Mean Latency, Mean Weight, Max Latency, and Number of serviced Jobs, for 10041 total serviced packets, in 178 cycles.

Queue	Mean Latency	Mean Weight	Max Latency	Number of Jobs
0	5727.28176100629	98.3012578616352	6986	1590
1	5728.09986595174	98.5764075067024	6281	1492
2	5722.24184397163	98.2170212765957	6275	1410
3	5723.66559485531	99.1744372990354	6278	1244
4	5720.85227272727	98.6298701298701	6281	1232
5	5728.53405017921	98.5779569892473	6283	1116
6	5719.83996212121	98.4611742424242	6278	1056
7	5724.15427302997	98.2297447280799	6281	901

Table 6.15-1 Each time a packet is dequeued then a new packet is enqueued in the same queue. Average packet weight 100, minimum queue weight 650, maximum queue weight 1000.

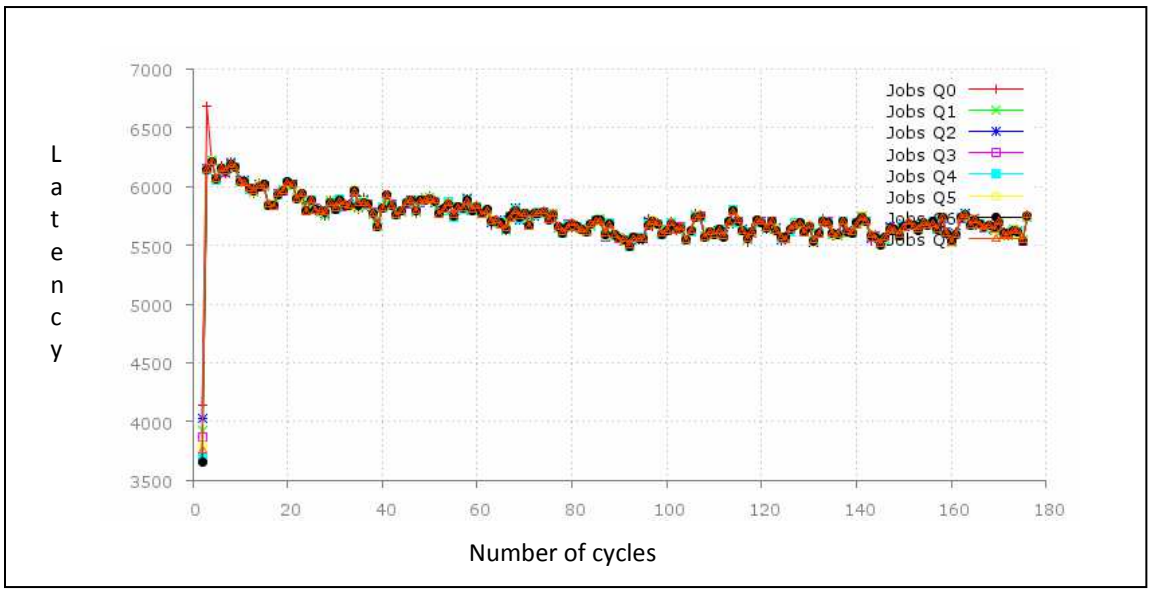

Figure 6.15-1 Latency for each time a packet is dequeued then a new packet is enqueued in the same queue. Average packet weight 100, minimum queue weight 650, maximum queue weight 1000.

From the above table we can understand that in such case, the Mean Latency is approximately the same for all queues. Q0 has the maximum latency. This occurs in the first cycle so it isn't representative. Q0 has serviced most jobs, 1590, next is Q1 which has serviced 1492 jobs, and so on. Q7 has serviced fewer jobs, 901. So Q0 has the best performance followed by the other queues.

In comparison with scenario 13, the only difference is in the weights of the queues, since the packets have the same Mean Weight, which is 100. As a result, in scenario 14 Q0, Q1, Q2, Q3 have served fewer jobs than scenario 13. Moreover Q4, Q5, Q6, Q7 have served more jobs than in scenario 13. Such a small decrease in the weight of queues has result in slight changes in the total number of serviced packets between neighboring tails.

6.16 Scenario 16. Average packet weight 50, minimum queue weight 650, maximum queue weight 1000. Each time a packet is dequeued then a new packet is enqueued in the same queue.

As in the previous scenario, the weights of the queues have been assigned as follows: Q0:1000, Q1:950, Q2:900, Q3:850, Q4:800, Q5:750, Q6:700 and Q7:650. But in this example the average weights of packets are 50. Table 6.16-1 shows the Mean Latency, Mean Weight, Max Latency, and the Number of serviced Jobs, for 10041 total serviced packets, in 82 cycles.

Queue	Mean Latency	Mean Weight	Max Latency	Number of Jobs
0	6172.39896707553	48.5842479018722	7090	1549
1	6165.38143631436	48.579945799458	6708	1476
2	6164.97780959198	48.4874731567645	6716	1397
3	6163.29543634908	48.8783026421137	6697	1249
4	6163.16530944625	48.4771986970684	6690	1228
5	6160.36271808999	49.0982552800735	6686	1089
6	6157.75609756098	48.4634146341463	6662	1066
7	6155.9179331307	48.2330293819656	6657	987

Table 6.16-1 Each time a packet is dequeued then a new packet is enqueued in the same queue. Average packet weight 50, minimum queue weight 650, maximum queue weight 1000.

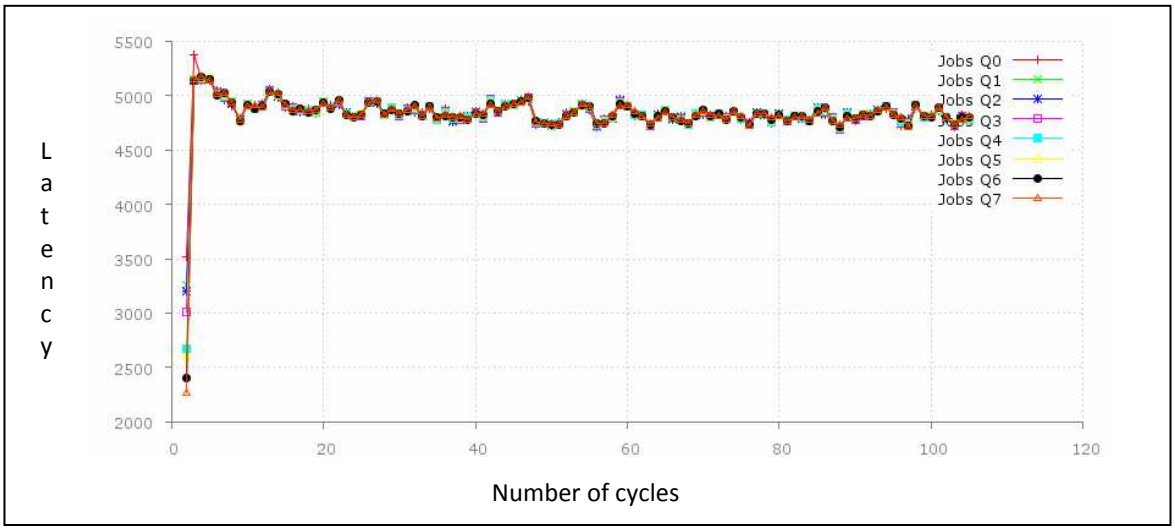

Figure 6.16-1 Latency for each time a packet is dequeued then a new packet is enqueued in the same queue. Average packet weight 50, minimum queue weight 650, maximum queue weight 1000.

From table 6.16-1 we can figure out that in such case, the latency is Q0 has the maximum Mean Latency and Q7 the minimum, but the difference between them is only 17 quanta. This occurs in the first cycle so it isn't representative. Q0 has serviced most jobs, 1549, next is Q1 which has serviced 1476 jobs, and so on. Q7 has serviced fewer jobs, 987. So Q0 has the best performance followed by the other queues.

In comparison with scenario 15, the only difference is in the weights of the packets, which is 50, whereas in scenario 15 it is 100. In both cases the number of serviced job is almost the same. But in scenario 15 the Mean Latency is less. Therefore larger weight of packet accrues better Mean Latency.

In comparison with scenario 14 the only difference is in the weights of the queues, since the packets have the same Mean Weight, which is 50. As a result, in scenario 16 Q0, Q1, Q2, Q3 have served fewer jobs than scenario 14. Also Q4, Q5, Q6, Q7 have served more jobs than scenario 14. So a small decrease in the weight of the queues has as a result the small changes in the total number of serviced packets between neighboring tails.

7. Conclusions and Future extension

In comparison to algorithms presented in related works, the algorithm now presented resembles Weighted Round Robin and Deficit Round Robin. However, both algorithms have been recommended for use in computer networks. WRR has been proposed for asynchronous transfer mode (ATM) networks and DRR for servicing queues in a router (or gateway). A necessary condition for WRR is the fixed size of packet. Each packet needs to have the same size in all queues. In contrast, the proposed algorithm is independent of the packet size. Also DRR services packets of different size but at the same time it services all packets together. Our algorithm, however, services one packet from a queue at a time. But in total, more than one packet will be served in proportion to queue weight. The enqueue time of serviced packets is also unclear for WRR and DRR. The above algorithms served all the packets independent of the enqueue time. For reason of fairness, though, the proposed algorithm serves only the packets that have been queued before a new cycle has started.

The algorithm combines different algorithmic techniques. Each separate queue is a queue FIFO. It doesn't require a sorted priority queue. So it is easy to be implemented and at the same time no computing power for classification is wasted. There is no prioritization which means that each process may eventually be completed, therefore, no starvation. All queues are organized as a multi-queue technique. Each queue has different total weight. Hence the total number of serviced job is proportional to the total weight for each queue. In a cycle the maximum total weight that can be served is at most equal to the sum of the weights of queues. For reasons of fairness the scheduler serves only the packets that have been enqueued when the cycle started. New packets enqueuing after the cycle is started will serve in the next cycle. When a new cycle started the first queue (the queue with the maximum weight) isn't examined first but the control continues from the next queue where it left off the previous cycle. Otherwise the lower weight tails will be served too late. Admittedly this favors the queues with lower weight in case of simultaneously enqueuing of new packets to all queues or enqueuing new packets just before the start of the new cycle. However, in a real time system this is rarely case, since in such systems the enqueue of a packet is continuously performed and furthermore not at the same time for all queues.

The queue with the highest weight serves larger number of packets. The queue with the lowest weight serves less number of packets. But the tail with the largest weight has a greater latency per cycle and the queue with less weight has less latency if the packets are enqueued just before the new cycle starts. If the weight of the packet (*packet* w_i) is smaller the difference in the average latency between neighboring queues is more balanced. If the packets are enqueued during the cycle, then the queue with largest weight has less mean latency. Also less weight of packets has as a result more packets to be serviced in the queue per cycle. Moreover the increase of the weight of the tail has as a result greater number of serviced jobs.

If in the tails are enqueued packets weighing more than the weight of a queue, then there is bottleneck. The problem is most acute in the lower weight queues where major service delays occur. If the weights packets follow a normal distribution and the enqueued packets have less weight than the weight of the queues the algorithm behaves like WRR with small differences in the average latency and the total number of performing packet per queue.

The timing of the enqeuing of packets is important to the performance of the algorithm. If new packets are enqueued at the beginning of the cycle they will serve in the next cycle so it will have the greatest latency. The ideal insertions of new packets take place just before the start of the new cycle, something hard in real-time systems. Moreover the selection of the queue for the enqueue of packets affects the performance of the algorithm. If there are several packets to serve, it is better to select heavy queues. If we are interested in the execution time of a process it is best queues to select with less weight.

Depending on the policy we want to follow we can make various modifications to the algorithm. In our algorithm the new enqueuing packets are not served during the same cycle. For reasons of fairness they will be served in the next cycle. Another idea is always to serve the new packets until the permitted weight of the queue, and not to wait for the next cycle. Another aspect is to serve the new packets until the queue weight; but if the queue becomes empty during the cycle without achieving the maximum serviced weight and then new packets are enqueued, these new packets will be served in the next cycle. This is easily done in our algorithm if the control of the field *order* is not to take place. Another

modification, similar to deficit round robin, is the remaining weight that is not served during the cycle to be added to the weight of the next cycle. However, in such case, the maximum serviced weight will be not fixed in a cycle.

References

[1] Rogers, P. (2013). Heterogeneous system architecture overview. In *Hot Chips*(Vol. 25).

[2] Rajput, I. S., & Gupta, D. (2012). A priority based round robin CPU scheduling algorithm for real time systems. *International Journal of Innovations in Engineering and Technology*, *1*(3), 1-11.

[3] Kinsy, M., & Devadas, S. (2014, September). Algorithms for scheduling task-based applications onto heterogeneous many-core architectures. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE* (pp. 1-6). IEEE.

[4] Kyriazis, G. (2012). Heterogeneous system architecture: A technical review.*AMD Fusion Developer Summit*.

[5] Chitlur, N., Srinivasa, G., Hahn, S., Gupta, P. K., Reddy, D., Koufaty, D., & Iyer, R. (2012, February). QuickIA: Exploring heterogeneous architectures on real prototypes. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on* (pp. 1-8). IEEE.

[6] ARM big.LITTLE, 2015, [online] available from: < https://en.wikipedia.org/wiki/ARM_big.LITTLE> [accessed 12/10/2015]

[7] Thompson, C. J., Hahn, S., & Oskin, M. (2002, November). Using modern graphics architectures for general-purpose computing: a framework and analysis. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture* (pp. 306-317). IEEE Computer Society Press.

[8] Lee, H., Faruque, A., & Abdullah, M. (2014, March). GPU-EvR: Run-time event based real-time scheduling framework on GPGPU platform. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014* (pp. 1-6). IEEE.

[9] Stone, J. E., Gohara, D., & Shi, G. (2010). OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, *12*(1-3), 66-73.

[10] Fang, J., Varbanescu, A. L., & Sips, H. (2011, September). A comprehensive performance comparison of CUDA and OpenCL. In *Parallel Processing (ICPP), 2011 International Conference on* (pp. 216-225). IEEE.

[11] Karimi, K., Dickson, N. G., & Hamze, F. (2010). A performance comparison of CUDA and OpenCL. *arXiv preprint arXiv:1005.2581*.

[12] Srinivasan, A. (2003). *Efficient and flexible fair scheduling of real-time tasks on multiprocessors* (Doctoral dissertation, University of North Carolina at Chapel Hill).

[13] Kaladevi M, Phil M. &.Sathiyabama S, (2010) “A Comparative Study of Scheduling Algorithms for Real Time Task”, *International Journal of Advances in Science and Technology*, (Vol. 1, No. 4).

[14] Sirohi, A., Pratap, A., & Aggarwal, M. (2014). Improvised Round Robin (CPU) Scheduling Algorithm. *International Journal of Computer Applications*, *99*(18), 40-43.

[15] Di Francesco, P., & Sweden, V. (2012). *Design and implementation of a MLFQ scheduler for the Bacula backup software* (Doctoral dissertation, Master thesis in Global Software Engineering).

[16] Lamminen, O. P. (2007). *Implementation and performance analysis of a delay based packet scheduling algorithm for an embedded open source router* (Doctoral dissertation, Helsinki University of Technology).

[17] Abraham Silberschatz, P. B. Galvin, G. Gagne, (2005), Operating System Concepts,7th edition, John Wiley & Sons.

[18] Katevenis, M., Sidiropoulos, S., & Courcoubetis, C. (1991). Weighted round-robin cell multiplexing in a general-purpose ATM switch chip. *Selected Areas in Communications, IEEE Journal on*, *9*(8), 1265-1279.

[19] Shreedhar, M., & Varghese, G. (1995, August). “E cient fair queuing using de cit round robin", In *Proceedings of the ACM SIGCOMM* (Vol. 95).

[20] Kinsy, M., & Devadas, S. (2014, September). Algorithms for scheduling task-based applications onto heterogeneous many-core architectures. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE* (pp. 1-6). IEEE.

[21] Arabnejad, H., & Barbosa, J. G. (2014). List scheduling algorithm for heterogeneous systems by an optimistic cost table. *Parallel and Distributed Systems, IEEE Transactions on*, *25*(3), 682-694.

[22] Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., & Planas, J. (2011). Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, *21*(02), 173-193.

[23] Chronaki, K., Rico, A., Badia, R. M., Ayguadé, E., Labarta, J., & Valero, M. (2015). Criticality-Aware Dynamic Task Scheduling for Heterogeneous Architectures

[24] Shetti, K. R., Fahmy, S., & Bretschneider, T. (2013, December). Optimization of the HEFT algorithm for a CPU-GPU environment. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2013 International Conference on* (pp. 212-218). IEEE.

[25] Topcuoglu, H., Hariri, S., & Wu, M. Y. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, *13*(3), 260-274.

[26] Canon, L. C., Jeannot, E., Sakellariou, R., & Zheng, W. (2008, January). Comparative evaluation of the robustness of dag scheduling heuristics. In *Grid Computing* (pp. 73-84). Springer US.

[27] Beaumont, O., Boudet, V., & Robert, Y. (2001). The iso-level scheduling heuristic for heterogeneous processors.