Contents

1. Introduction

In this chapter, our work is introduced and the goals of the thesis are defined.

The energy demands of data centers are continuously increasing over time. In 2014, data centers in the U.S. consumed an estimated 70 TWh, representing about 1.8% of total U.S. electricity consumption. It is expected to be 73 billion kWh by 2020 [SSS+]. Talking about the data centers in Germany, the electricity consumption of the data centers has increased by 3% to 12 TWh in 2015. Compared with other European countries, Germany has the largest data center market with a share of approximately 25% of European data center capacity [Hin]. Other than saving energy, a data center must provide a quality of service (QoS) as agreed upon in the service level agreement (SLA). Extensive research is being done to develop technologies that reduce the power consumption of the data centers while meeting the terms of SLA. This work focuses on one such approach called utilization based *Dynamic Voltage and Frequency Scaling* (DVFS). In DVFS, the frequency of the processor is adjusted proportionally to the work load. To study the behaviour of a DVFS enabled CPU, Markov chain models are used. Markov chains allow us to generate statistic simulation models of real world processes.

Many current processors support DVFS, we have used the servers from vendors Intel and AMD to evaluate the accuracy of Markov models. Linux kernel implements DVFS via CPUFreq. This work focuses on *Ondemand Governor* provided by CPUFreq. Ondemand governor checks the CPU usage statistics over last period and sets the CPU frequency accordingly. A recently published paper "Modelling Performance and Power Consumption of Utilization-based DVFS Using M/M/1 Queues" [BNdM16] is the base idea for this thesis. It will be referred as the base paper throughout the work.

This section mentions the respective goals and results achieved in this thesis. The first goal was to confirm the correctness of the adopted methodology in the base paper by performing similar experimental analysis as on a different (newer) hardware setup. It was noticed that current Intel processors use `intel_pstate` driver by default, hence the CPUFreq driver must be explicitly enabled to be able to use the CPUFreq's ondemand governor. The obtained results show that the suggested Continuous-Time Markov Chain (CTMC) of M/M/1-FCFS queuing system and Discrete-Time Markov Chain (DTMC) of DVFS enabled CPU with ondemand governor work very well with the new hardware, in spite of the differences in versions of ondemand governor modelled in the DTMC and the version installed on the proces-

sor. Measurements from two different architectures were measured and compared to the modelled results to prove that the implementation of M/M/1-FCFS queuing system mentioned in this thesis is architecture independent.

We have developed a tool that is used to measure performance metrics of a processor, and is referred as *M/M/1 software*. This software mimics the M/M/1-FCFC queuing system. The performance metrics it measures include service time, service rate, mean response time, utilization caused by the subjected workload. We have also attempted to suggest a new model for the ondemand governor based on empirical analysis of frequency statistics. This model predicts the service rate of a DVFS enabled processor.

The input parameters such as arrival rate, service rate and the size of arrival queue play a crucial role in modelling the M/M/1 queuing system. These parameters must be chosen in a way that the system is in equilibrium. A system is in equilibrium when the utilization of the processor is less than or equal to 1. It was made sure that no jobs were rejected by the server to maintain the consistency of the results obtained. Hence, all jobs were processed to get theoretically correct values of mean service time and mean response time.

SHARPE (Symbolic Hierarchical Automated Reliability and Performance Evaluator) has been used to analyze the DTMC of DVFS-enabled CPU with ondemand governor and calculate the mean response time (MRT) and power consumption from the DTMC model.

This thesis contributes an improved Java implementation of a client-server setup representing M/M/1-FCFS queuing system. It precisely calculates the mean service time and mean response time for the jobs received in a given duration of time. Second contribution is the comparison study between *CTMC* representation of M/M/1-FCFS, M/M/1 software and *DTMC* of DVFS enabled processor. MRT and power consumption were measured using M/M/1 software and compared with modelled values obtained from the *CTMC* and the *DTMC* to confirm their accuracy. Third contribution is the study of the behaviour of *Ondemand Governor* at different utilization levels of the processor. Last contribution is to develop a methodology to calculate the steady state probability of each frequency by using the data provided by frequency statistics of ondemand governor. Using these steady state probabilities we propose a NewDTMC model as discussed in Section 4.4.

2. Background

In this chapter, different tools and technologies used in this work are discussed. At first, the detailed description of multi-core processors and available frequencies are discussed. The next important topic is the CPUFreq driver and the various governors provided by the Linux kernel. And lastly the tools like cpufrequtils, SHARPE, Eureqa and LMG power analyzer will be introduced. These tools and technologies form the backbone of this research.

2.1. Multi-core Processors

All the experiments in this work have been performed on multi-core processors. This section explains the basics of multi-core processors. A processor is a component that interprets and implements program instructions. It is known as central processing unit (CPU). A multi-core processor is a single piece of hardware with multiple independent CPUs called cores on a single chip carrier or die. The user can control if the assigned task should run on single core or be distributed among multiple cores. Linux command "taskset" provides the functionality of pinning a task to a particular core. Hyper-threading[1] has been deactivated for all cores. Each core is completely devoted to only one task. Also turbo boost setting has been disabled to make sure that processor does not run above its maximum operating frequency, thus preventing dynamic changes in processor's clock rate. For the purpose of modelling it is very important that we have full control over the functioning of the processor.

2.1.1. Frequency

Clock rate commonly known as frequency represents the speed (clock cycles per second) of a processor. Each processor comes with a predefined set of frequencies that it can attain. For example the Intel Xeon(R) CPU E5-2650 v2 @ 2.60GHz processor used in this research has 16 possible frequency settings available ranging from minimum 1.2 GHz to maximum 2.6 GHz. Another CPU used is this research is an AMD Opteron(tm) 4180 processor. It supports five frequencies: 0.8GHz,

[1] Hyper-Threading Technology makes a single physical processor appear as two logical processors; the physical execution resources are shared and the architecture state is duplicated for the two logical processors. [DM02]

1.3GHz, 1.8GHz, 2.3GHz and 2.6GHz. Higher the frequency of a processor, higher power consumption.

2.1.2. C-States

C-states are power modes that a processor enters to reduce its power consumption. During very low processor utilization, the CPU automatically switches to low power modes [Tor08]. These modes are represented by C-states. There are six C-states ranging from C0 to C6. C0 stands for operating state where all CPUs are fully turned on. States C1 to C3 work by cutting clock signals used inside the CPU while C4 to C6 work reducing CPU voltage. C0 consumes the most power and C6 the least [Kid14]. To study and model the behaviour of ondemand governor, C-states of the processor were disabled, so that it would not interfere with the functioning of ondemand governor.

2.2. Governor

Governors are basically the decision makers that decide the frequency at which the processor run. The frequency change can be triggered by various events such as utilization level of the processor or manually by userspace programs. A policy consists of frequency limits (policy→min,max) and the governor decides what target frequency is set within the limits of the policy. CPUFreq is a driver implemented by the Linux kernel. It is a CPU frequency scaling algorithm that can adjust CPU frequency dynamically, depending on which governor has been set by the user. [Bro]

Following are the governors provided by the Linux kernel:

2.2.1. Types of Governors

Performance governor locks the CPU frequency to maximum possible.

Powersave governor locks the CPU frequency to the lowest frequency available.

Userspace governor allows the user to decide as to which available frequency must be set. The user can set the frequency inside the "`scaling_setspeed`" file.

Ondemand governor is the main focus of this work. This governor sets the frequency of the CPU based on its estimated load. The linux kernel scheduler triggers load estimation after fixed interval of time, CPUFreq monitors the usage statistics over the last interval and governor sets CPU frequency accordingly. Ondemand governor provides various parameters that can be set by the users. The following parameters are relevant to this work: `Sampling_rate` defines how often the kernel

monitors CPU usage statistics and takes a decision regarding frequency switching. `Sampling_rate_min` defines the smallest possible `sampling_rate` δ, which can vary depending on the hardware. `Sampling_rate` is measured in microseconds. `Up_threshold` u is the percentage of CPU utilization that is needed to trigger a switch to the highest frequency. In case the CPU utilization is below this threshold, the older version-2.6.9 of ondemand governor decreases the CPU frequency step by step until it reaches the lowest frequency. However, in the newer version-2.6.16, ondemand has been modified to more aggressive frequency down-scaling. The governor intelligently takes a decision to jump to frequency that keeps the CPU 80% busy[VP]. This work uses the new version of ondemand governor, the results obtained in Chapter 5 confirm that this added feature has no significant impact on the performance of the processors.

Conservative governor works on the similar principle as ondemand governor. The only difference is that it gradually increases or decreases the CPU frequency.

2.2.2. Intel P-States

Intel P-state is also a power scaling driver like CPUFreq, it comes by default with newer Intel CPUs. CPUFreq is a module that can be loaded or unloaded, but P-states is built into the CPU. This is the latest scaling driver in the market. According to [web], `Intel_pstate` supports only powersave and performance policies. Intel P-state is a governor and hardware driver combined in one. The "performance" policy always picks the highest pstate, but the "powersave" policy is more like the ondemand governor, it attempts to balance performance and energy saving. Pstate driver decides the next P-state based on the requested cpufreq policy- powersave or performance, but if the processor is capable of selecting the next P-state internally then the responsibility is given to the processor [web].

2.2.3. Manipulating Governors

"cpufrequtils" is a tool available in most Linux distribution. It allows us to tweak and control CPUFreq settings from the command line. "cpufreq-set" comes with various useful options such as -c (to choose the CPU where the setting should apply), -g (to set a new governor), -d (minimum CPU frequency governor may select), -u (maximum CPU frequency governor may select), -f (to set a specific frequency in case userspace governor is loaded).

Next, we show how we can use these commands to read and manipulate the governors.

- To read CPU architecture information: *lscpu*
- To view current frequency of all cores, where n represents the CPU:

sudo watch cat /sys/devices/system/cpu/cpu{0..n}/cpufreq/cpuinfo_cur_freq

- To view the available governors:

  sudo watch cat /sys/devices/system/cpu/cpu{0..n}/cpufreq/scaling_available_governors

- To view the governor that is currently active:

  sudo watch cat /sys/devices/system/cpu/cpu{0..n}/cpufreq/scaling_governor

- To view the maximum operating frequency:

  sudo watch cat /sys/devices/system/cpu/cpu{0..n}/cpufreq/cpuinfo_max_freq

- To view minimum operating frequency:

  sudo watch cat /sys/devices/system/cpu/cpu{0..n}/cpufreq/cpuinfo_min_freq

- To view all available frequencies:

  sudo watch cat /sys/devices/system/cpu/cpu{0..n}/cpufreq/scaling_available_frequencies

- To view the set frequency when userspace governor is active:

  sudo watch cat /sys/devices/system/cpu/cpu{0..n}/cpufreq/scaling_setspeed

- To view the cpufreq driver that is used to set the frequency of the CPU:

  sudo watch cat /sys/devices/system/cpu/cpu{0..n}/cpufreq/scaling_driver

- To change the governor of core number n to performance:

  sudo cpufreq-set -c n -g performance

- To set the speed/frequency of the processor manually (Userspace governor):

  sudo cpufreq-set -c n -f 2100000

- To set threshold for ondemand governor:

  echo "95" > /sys/devices/system/cpu/cpufreq/ondemand/up_threshold

- To set the sampling rate of ondemand governor:

  echo "120000" > /sys/devices/system/cpu/cpufreq/ondemand/sampling_rate

- To measure the utilization of the processor :

  mpstat -P (Core Id or ALL) (Interval) (No. Of outputs)

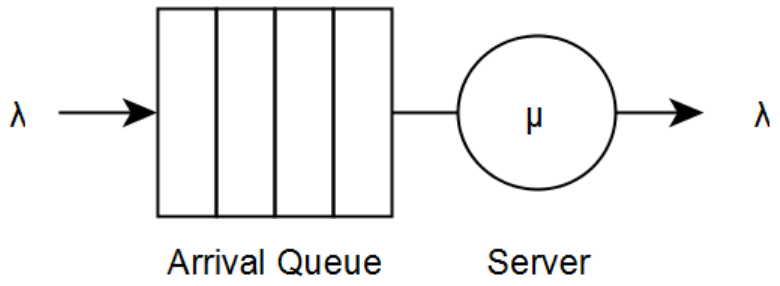

Figure 2.1.: M/M/1 Queuing system

2.3. M/M/1-FCFS

Figure 2.1 represents the high level model M/M/1-FCFS of a queuing system. The name of the model written in Kendall's notation A/B/m - queuing discipline, where 'A': arrival process, 'B': service process and 'm': number of servers. 'A'and 'B 'are represented with symbol 'M', where 'M 'stands for Markov process and FCFS for First-Come-First-Serve. A Markov process is a process which satisfies "memoryless property" i.e. the future of the process does not depend on past states and the age of the current state. Both arrival and service processes are stochastic in nature. The arrival process is a Poisson process with rate parameter λ, it represents mean number of job arrivals per unit time. The inter-arrival times of Poisson process are exponentially distributed with mean $1/\lambda$. The service process defined how the jobs are processed inside the server. A Markovian service process has exponentially distributed service time, such that jobs are processed at the rate of μ jobs per second. For the system to be stable, the arrival rate λ must be less than service rate μ. M/M/1 queuing system can be mathematically analyzed to calculate performance measures such as,

$$\pi_k = P[\text{there are k jobs in the system}], \tag{2.1}$$

$$\text{Utilization } \rho = \frac{\lambda}{\mu} = 1 - \pi_0, \tag{2.2}$$

$$\text{Response Time } T = \frac{1/\mu}{1-\rho} \tag{2.3}$$

$$\text{Power } P = (1 - \frac{\lambda}{\mu}) * P_{idle} + \frac{\lambda}{\mu} * P_{loaded} \tag{2.4}$$

In Figure 2.2, CTMC of M/M/1-FCFS queuing system is presented. It is also known as simple birth-death process. The jobs enter the system with rate λ and get serviced with rate μ. The system is called to be in steady state, when its behaviour is time independent. The steady state probabilities for the CTMC can be calculated as per Equation 2.1. Refer book [GBT06] for detailed explanation.

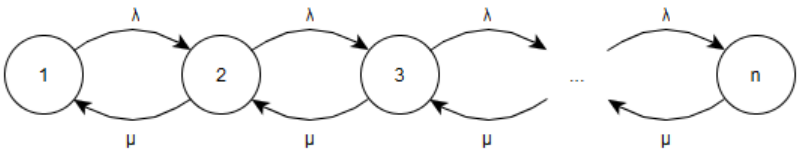

Figure 2.2.: CTMC of M/M/1 Queuing system

2.4. Devices and Tools used in our work

To measure the power consumed by the DVFS enabled processor on application of Markovian workload, we used LMG500 power meter. This is a power analyzing hardware that is used to capture the real time power measurements of a processor. It takes continuous samples at the rate of 2 samples/second and claims accuracy of 0.025 percent.

We also used Eureqa for generating equations that predict steady state probabilities using the data obtained from M/M/1 software as mentioned in Section 4.4.3. Eureqa is a artificial intelligence powered modelling engine. This tool performs evolutionary search on data to generate equations, hence allowing us to find relationship between different sets of data.

SHARPE stands for Symbolic Hierarchical Automated Reliability and Performance Evaluator. It provides with specification language and solutions for various queuing networks. We have used this tool to analyze the markov chain models mentioned in Chapter 4.

3. Implementation

This chapter describes the implementation of a software implementing M/M/1-FCFS queuing system. We refer to it as the *M/M/1 software*. We have also validated the correctness of the software in Section 3.3.

The main aim of this research is to model performance and power consumption of a "DVFS enabled processor" using M/M/1 queues. The reason for using M/M/1-FCFS queuing system is that Markov chains provide us with closed form solution to calculate performance metrics such as mean response time of the system and power consumption. The base paper [BNdM16] describes in detail how M/M/1 queues are suitable for modelling performance and power of processor. For the purpose of our research we assume that this approach is an accurate representation of real world scenario. We now explain M/M/1 queuing system. Such a system must satisfy the following basic properties. The arrival process should be a Poisson process i.e. the inter-arrival time between the jobs is a sequence independent random variable, each having exponential distribution with rate parameter $1/\lambda$, therefore leading to the mean *arrival rate* of λ(jobs/sec). The service times of the jobs should be exponentially distributed with rate parameter μ.

Figure 3.1 gives a graphical representation of how exponentially distributed events happen. The events appear to occur in an irregular pattern, although over a period of time they are exponentially distributed with rate parameter λ. The jobs are served at single service station in a first come first serve fashion. In this thesis, we have implemented a software that replicates M/M/1-FCFS queuing system. It records performance metrics such as service time, mean service rate and mean response time of the processor for the given workload. It also records the CPU frequency statistics during each run of the software. The frequency statistics include "time_in_state" and "trans_table" as mentioned in [Pal]. However, to measure the power consumption we use an external power meter. Each run of the software is referred to as a client-server session that lasts for a specified amount of time.

This paragraph describes the basic components of the M/M/1 software. It consists of two components, namely client and server, with an externally connected power meter. A client generates job packets with an exponentially distributed *arrival rate* at the server. It assigns every job with a *repeat* value, which is exponentially distributed as well. The *repeat* value determines the size of each job, higher *repeat* value implies that higher workload is generated by the job. The client generates reset packets at the start and end of a session. The server receives the job packets and

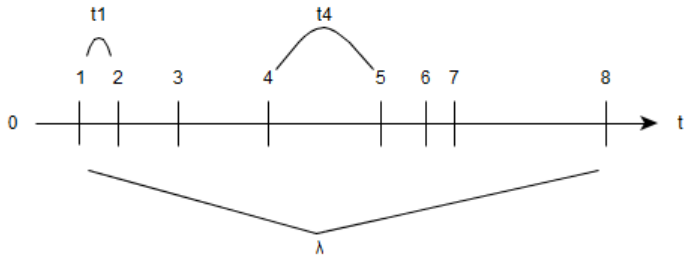

Figure 3.1.: Exponential Distribution

processes them one by one. Each job performs CPU intensive tasks and is pinned to a single core. Since the job's repeat value is exponentially distributed, the service rate is also exponentially distributed with mean rate μ. Each session is long enough to achieve a mean *arrival rate* as close as possible to user defined λ. With the execution of each job, the software measures the performance metrics of each job and stores the job data in a data structure. At the end of a client-server session, individual job data is combined to calculate mean performance metrics. A power meter attached to the server measures the power consumed in each session. The server also measures the frequency statistics, a snapshot of the system statistics is taken in the starting and the end of every session. The first snapshot is subtracted from the last one to calculate the statistics of that particular session, this is necessary because each snapshot contains the frequency statistics from the time of the operating system was installed. Another feature of the server is that it implements an algorithm to directly measure the steady state probability of the DTMC mentioned in Figure 5 of base paper [BNdM16]. The basic principle of this algorithm is that as M/M/1 executes only one job at a time, the state of the system changes only if a job arrives or leaves the system. The M/M/1 software captures the state of the system at these two instances and stores the number of times each state occurred. Hence, allowing us to calculate the probability of a particular state.

3.1. Implementation of M/M/1 Software

This section includes detailed description on how we implemented each component of M/M/1 software. It consists of two major parts: Client and Server as shown in Figure 3.2

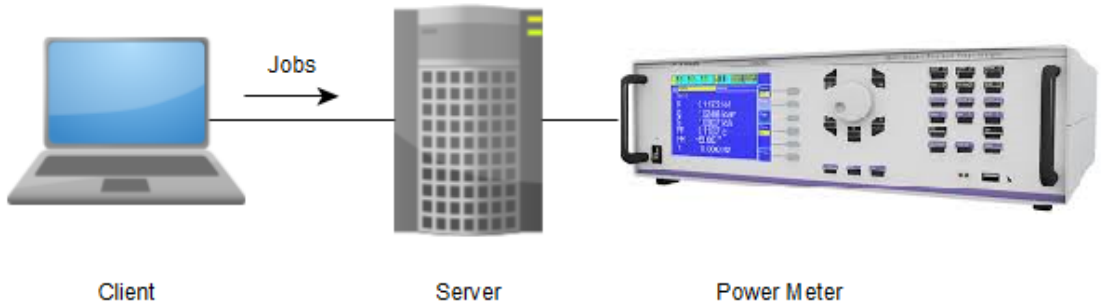

Figure 3.2.: Client and Server implementing M/M/1 queuing system

3.1.1. Client

The client is responsible for sending job packets to a server with exponentially distributed *arrival rate* λ, where the size of each job packet is exponentially distributed with rate *repeat*. The two classes that form the client are called Distribution.java and LoadGenerator.java. Figure 3.3 shows the client's class diagram.

Load generator accepts three values from the user, namely

- *arrival rate* - number of jobs arriving per second λ
- *duration* - duration for which LoadGenerator generates jobs
- *repeat* - determines the size of jobs

The `LoadGenerator` class initiates a session with the server by broadcasting a reset packet via the `resetBroadcastSocket`. The first packet called RESET1 is sent to `ServerIPAddress` at `resetListenerPort` to mark the start of a session and it signals the server about the beginning of transmission of job/work packets and to start the frequency statistics logging. Next, the work packets are sent to `jobListenerPort` via the `loadGeneratorSocket`. The jobs are sent with exponentially distributed inter-sending time with parameter $1/\lambda$ for the defined *duration*, so

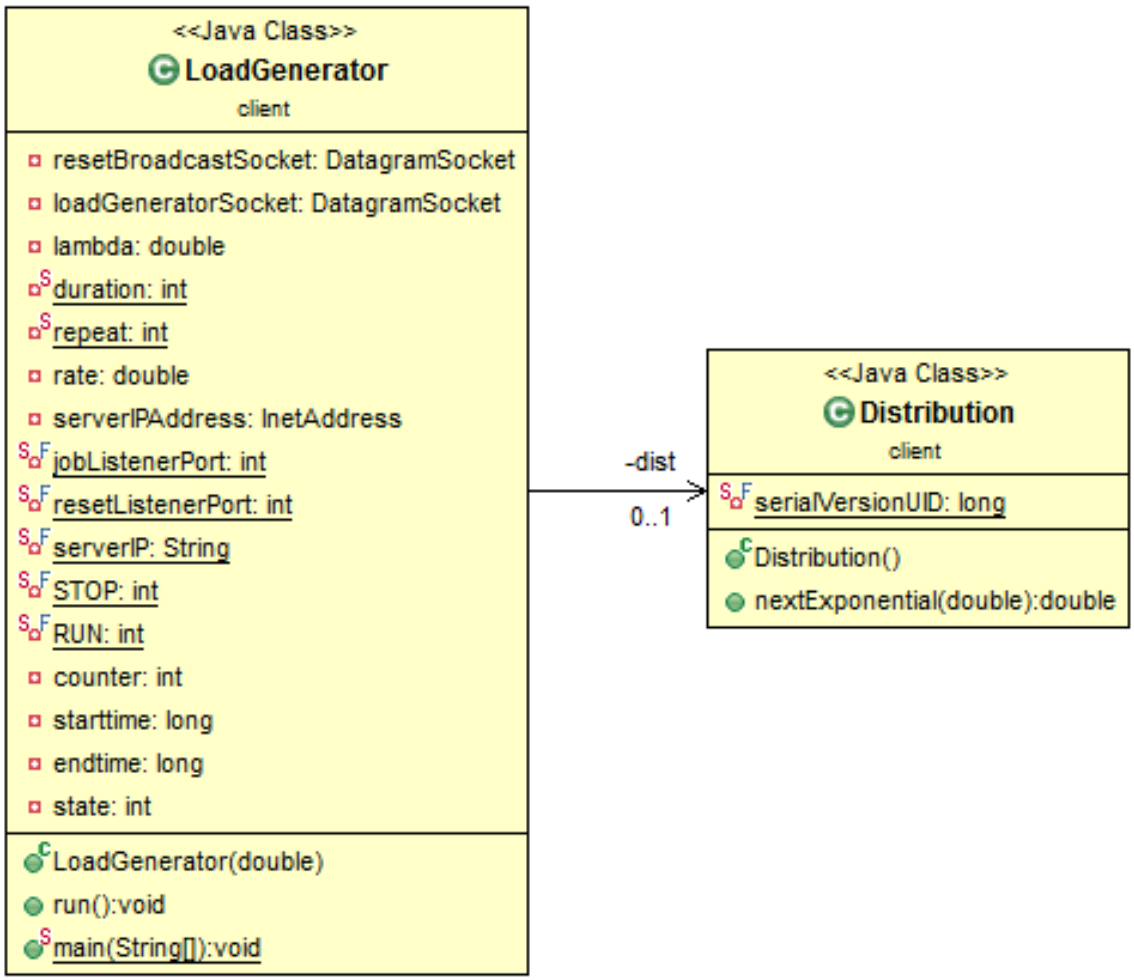

Figure 3.3.: Client class diagram

that the mean *arrival rate* of the jobs at the server is λ. These packets contain exponentially distributed integer value called *repeat* as their content. The *repeat* value is proportional to the service time of a job. Since the service times are exponentially distributed, it results in a service rate μ at the server. The `nextExponential` method of Distribution class is responsible for generating exponentially distributed inter-sending times and *repeat* values for the LoadGenerator class. At the end of the session, one last packet called RESET2 is sent right after the last job. It marks the end of transmission process at the server. Indicating the server to initiate metric calculation, stop frequency statistics logging and resets all the variables at the server to zero. After that the server is ready for the next session. At the end of each session, jobs sent per second are calculated with the help of `counter`, `starttime` and `endtime`. The variable `state` takes two values "RUN" and "STOP", these states decide whether the LoadGenerator is allowed to send jobs to the server or not, RUN indicates 'yes' and STOP means 'no'.

3.1.2. Server

The server listens for packets sent by the load generator and processes them. It consists of several classes, we have divided the server into different parts depending on its functionality. The parts are Server Initialization, WorkPacket Processing, ResetPacket Processing, Workload Execution and Metric Calculation.

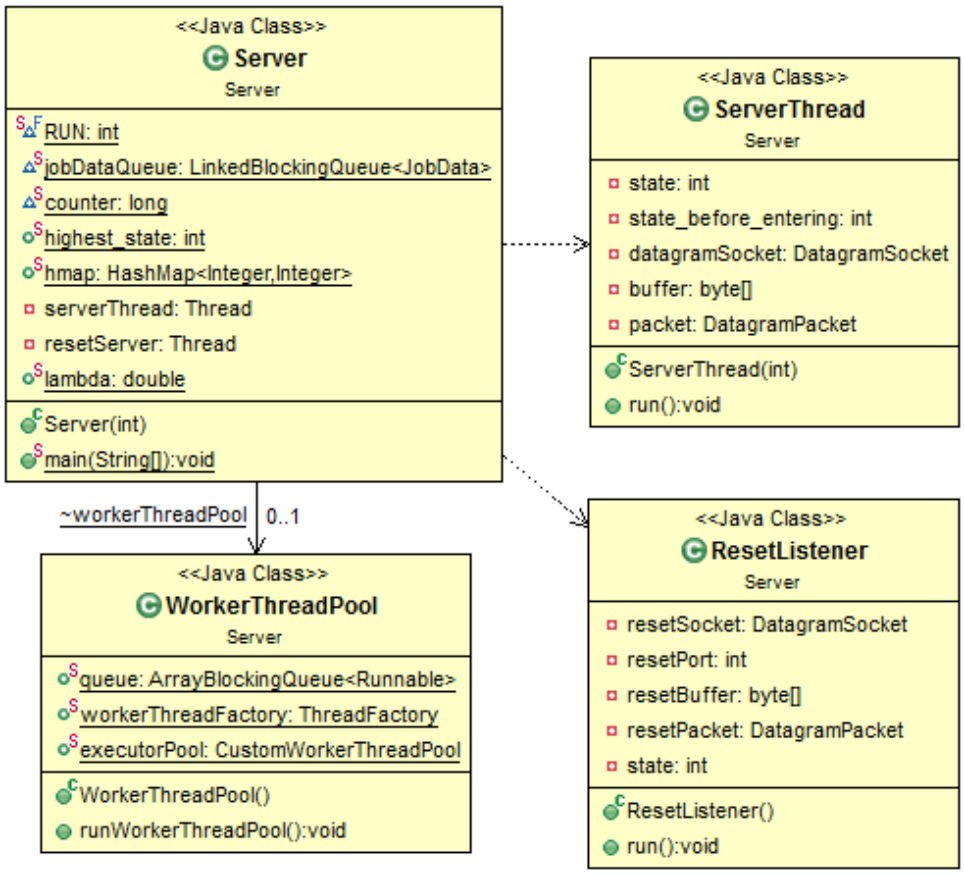

Figure 3.4.: Server Initialization class diagram

Server Initialization

Server starts with class Server.java, it is responsible for starting the three main threads, ServerThead, ResetListener and WorkerThreadPool. Figure 3.4 represents the class diagram for `server initialization`. The server also initializes a HashMap that is used to measure and calculate the steady state probabilities of DTMC mentioned in Figure 4.2. A jobDataQueue is created, class Job uses it to store service time, response time and *repeat* value for each job that is serviced by the server.

RESETPacket Processing

Figure 3.5 shows how RESET packets are processed by the server. The RESET packet is received by a DatagramSocket at class ResetListener and its contents are extracted. The content contains the name of the reset packet indicating whether it is the first (RESET1) or last (RESET2) reset packet. Based on the content, an instance of class ResetJob is created and added to the queue of WorkerThreadPool. A boolean flag is set at the ResetJob before adding it to WorkerThreadPool, the value of this flag determines whether it performs the tasks assigned to RESET1 (flag:true) or RESET2 (flag:false). RESET1 is the first packet to arrive in the system and its job is to initialize the hash-map values to zero and log the frequency statistics at the beginning of client-server session. RESET2 arrives last in the system. It has three main tasks, first to log frequency statistics, second to calculate performance metrics after the server finished processing all the jobs and third to set all the system queues and variables to zero.

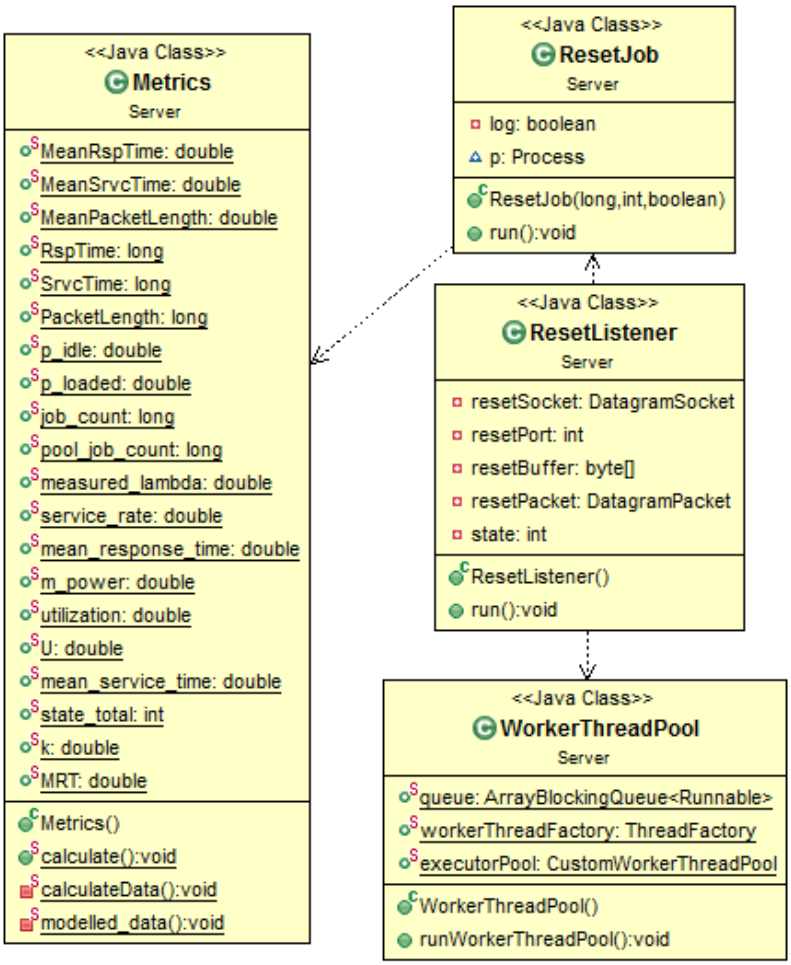

Figure 3.5.: ResetPacket processing class diagram

WorkPacket Processing

The workPacket represents the jobs that arrive at the server for processing. Figure 3.6 shows the class diagram of how a Job is processed. A datagramSocket is continuously listening to incoming job packets at class ServerThread. Once the packet arrives, the integer indicating the size of the job is extracted from it and an instance of Job class is created using this size and the current timestamp. Right before the job is passed to the WorkerThreadPool for execution, the state of the system is recorded in the hash-map named `hmap` in the Server. The state is computed by counting the total number of jobs in the system.

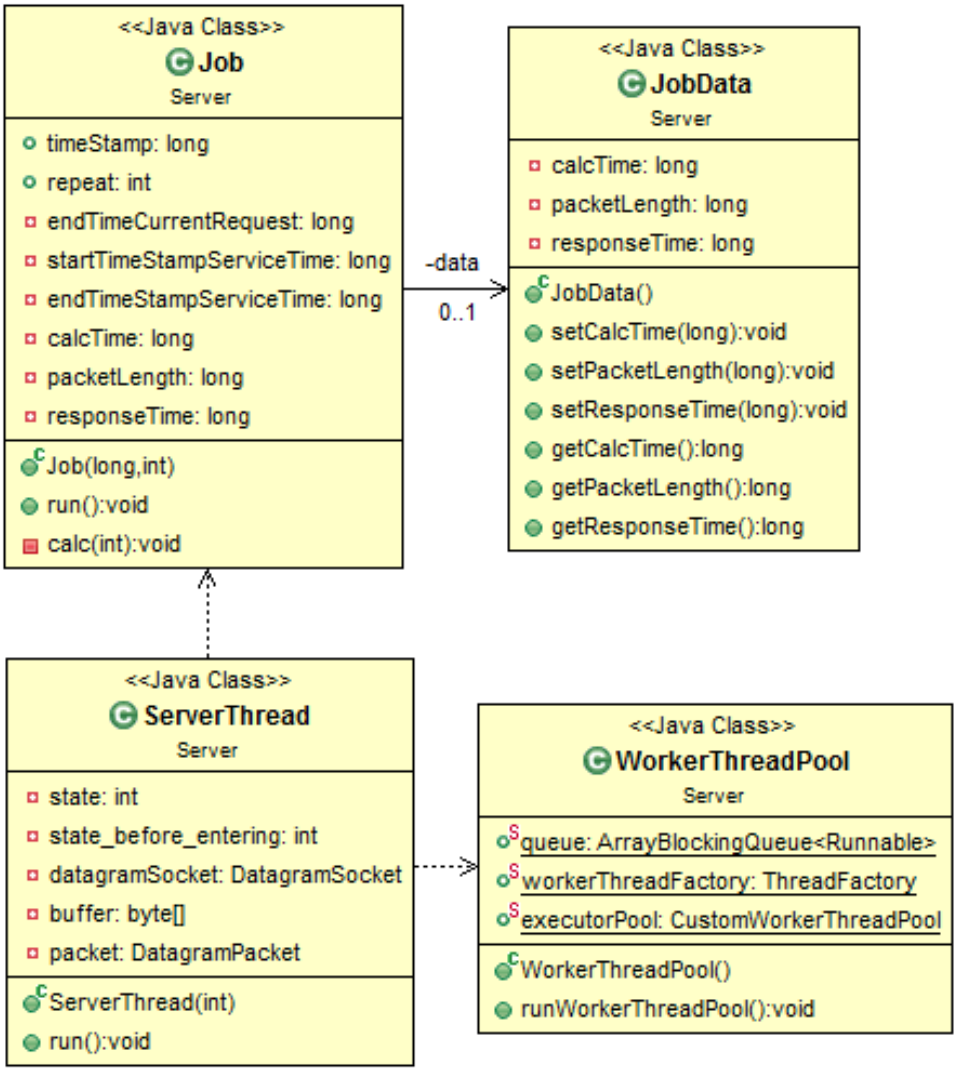

Figure 3.6.: WorkPacket processing class diagram

Workload Execution

As seen in the above sections, a Job and a ResetJob are given to WorkerThreadPool for execution by their respective listener classes, ServerThread and ResetListner. This section explains how exactly these jobs are processed by the WorkerThreadPool.

When a job is submitted to the queue of WorkerThreadPool, the pool takes one job at a time on first come first serve basis, and spawns a new thread for each job on a separate core. Each thread is an instance of the class `ThreadPoolExecutor` and it calls its protected overridable methods `beforeExecute()` and `afterExecute()` right before and after execution of each task. To process each job its `run()` method is executed. This method contains the CPU intensive workload that is executed as per the *repeat* value of each job. Once the job is finished, the `afterExecute()` method updates the hashmap value for the current state of the system. In case of a `ResetJob`, its `run()` method includes the tasks such as recording frequency statistics, initializing hash-map and setting the queues and variables of the server to zero. The tasks vary with the type of reset packet, RESET1 or RESET2.

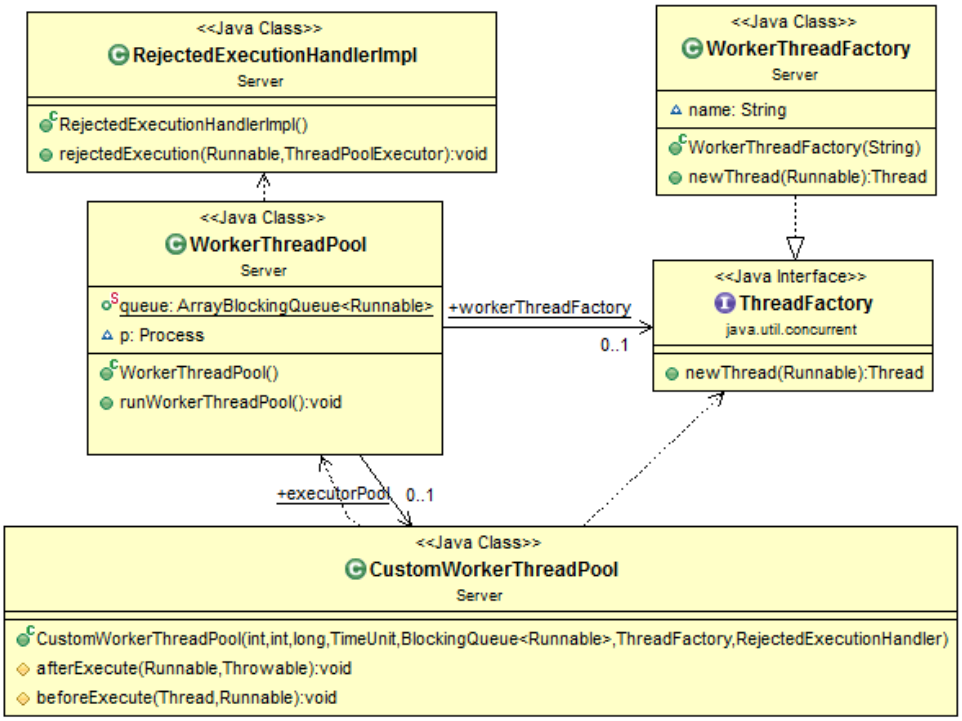

Figure 3.7.: Workload execution class diagram

3.1.3. M/M/1 Software Server Code

The code for server side of M/M/1 software consists of multiple classes. The java code for both client and server implementation has been included in the CD, that can be found at the back of this thesis under folder *M/M/1 Software*. This section is dedicated to explaining the role of each class in the server:

`Server`: The server runs as long as the value of `RUN` is set to 1. The server accepts λ from the user, this value represents the expected *arrival rate* of jobs coming from LoadGenerator.java. At first it creates a `jobDataQueue`, this queue is used by Job.java class to store performance metrics plus other data about each job, this data is later used by class `Metrics` to calculate mean service time and mean response time of the processor. Next it initializes a HashMap called `hmap` which records the frequency of each state (number of jobs in the system). At last it creates threads of classes ServerThread.java and ResetListener.java, respectively called `serverThread` and `resetServer`. It also calls the `runWorkerThreadPool` method of `WorkerThreadPool` class. The variable `counter` is used to count the number of jobs entering the system and `highest_state` is used to store the maximum number of jobs that were in the server during a client-server session.

`ServerThread`: The ServerThread class listens for incoming DatagramPackets on a DatagramSocket, as long as the `state` of the server remains 1. When a `packet` arrives, its contents are extracted from the `buffer`, the buffer contains the size of the job. Next, the state of the system is calculated and its corresponding frequency is incremented in the `hmap`. At last, a new runnable instance of Job class is created and given to WorkerThreadPool for execution via the method execute(Runnable).

`ResetListner`: The ResetListener listens on the `resetPort` for `resetPacket` using a `resetSocket`, as long as its `state` remains 1. The `resetBuffer` contains the name of the reset packet: RESET1 or RESET2. Next, a runnable instance ResetJob class is created, where the `log` flag is set to true for RESET1 and false for RESET2. In the end these jobs are passed to WorkerThreadPool for execution via the `execute(Runnable)` method.

`WorkerThreadPool`: The `WorkerThreadPool` is activated by the `Server` class. It creates an instance of class CustomWorkerThreadPool.java known as `executorPool`. It requires the following initial parameters:

```
int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit
unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
RejectedExecutionHandler handler.
```

If less than "corePoolSize" number of threads are running when a new job arrives, a new thread is created, else if the number of threads are more than "corePoolSize" and less "maximumPoolSize", a new thread is created only if the "WorkQueue" is full. The excess threads (threads created after corePool threads) if idle, are terminated

after "keepAliveTime" having specified "unit". Every new submitted job is stored inside `queue` and "workerThreadFactory" creates a new thread for executing the jobs. The RejectedExecutionHandler informs the user in case a job is rejected by the executorPool.

`Job`: The `Job` class defines the structure of work job arriving from the client. It has two parameters "timeStamp" and *repeat*. The `timeStamp` states the time the job was received by the server and *repeat* stands for the size of that job. When the job is executed, its `run` method is invoked, this method makes a call to `calc` method. The calc method contains the CPU intensive calculation which is repeated as per the size of the Job. The `calcTime` is the difference between `endTimeStampServiceTime` and `startTimeStampServiceTime` and `responseTime` is the difference between `endTimeCurrentRequest` and "timeStamp". The calctime , responseTime and packetLenght are stored inside the jobDataQueue initialized by Server.java.

`ResetJob`: The `ResetJob` is a subclass of `Job` with an extra boolean parameter called `log`, if set to true, it represents RESET1, else RESET2. RESET1 arrives in the beginning of a client-server session, its job is to initialize the `hmap` by setting the frequency of all the states to zero and take a snapshot of the frequency statistics of the core on which the work jobs are executed with the help of Process `p`. RESET2 is processed after all the work jobs are finished. It has three tasks: First, take a snapshot of frequency. Second, make a call to Metrics.java for performance metric calculation. Third, empty the jobDataQueue and all other system variables to zero.

`Metric`: This class is responsible for calculating the performance metrics mean service time, mean response time, service rate, utilization, power consumption and steady state probabilities. These values are stored inside the excel sheet document. There are three ways of calculating the above metrics. First method uses measurements stored in jobDataQueue that are processed in `calculateData`, such as RspTime, SrvcTime, PacketLength. The second method calculates steady state probabilities with the help of `hmap` which are then used to calculate mean number of jobs `k`, mean response time `MRT` and utilization `U` as mentioned in Section 4.2. This logic is executed inside method `calculate`. The third method used for calculating the power and performance metrics is calculated with the equations mentioned in Section 2.3. This logic is mentioned in method `modelled_data`.

`JobData`: This class is used to store and retrieve performance metric such as `calcTime`, `packetLength` and `responseTime` of each job packet that arrives at the server.

`CustomWorkerThreadPool`: This class inherits from `ThreadPoolExecutor` and overrides two of its methods called `afterExecute(Runnable, Throwable)` and `beforeExecute(Thread, Runnable)`. The afterexecute method is used to capture the state of the server and increment the corresponding frequency in`hmap`, after each job has finished execution. We will now discuss the shortcomings of initial

implementation, hence explaining the need for new and improved implementation.

3.2. Initial Implementation vs Improved M/M/1 software

Each time a job is sent by the client, it is received by the server on a datagram socket. The packet contains the size of the workload. The contents are extracted and then a job thread is created using this size and current timestamp. The above activities can be categorized as packet processing. At last the job thread is submitted to WorkerThreadPool for execution. The old implementation was performing the packet processing along with the job execution on the same core. Packet processing has a large noticeable overhead at higher arrival rates. This is inefficient as the core reaches unexpectedly high utilization levels caused due to this overhead.

The solution to this problem is to run the packet processing and job execution on two different cores. This has been implemented by pinning the server to a particular core and assigning the job execution to different one. Now we can study the 'core' processing the jobs in isolation, which is crucial for obtaining accurate service time and mean response time for each job.

The next issue was that the server abruptly shutdown after a fixed interval of time even if all jobs did not finish processing. Each client-server session lasts t minutes. This means that client sends job packets to server with rate λ for t minutes and stops. In these t minutes the server receives the jobs, adds them to its arrival queue and executes them in first come first serve order. The old implementation stopped the server at the end of t minutes without considering the completion of all jobs. The jobs that were still waiting in the queue to be processed were lost. To solve this problem, reset packets were integrated with the arrival queue. A reset packet was introduced inside the arrival queue after the last job packet arrived. The server processed the jobs until it encountered the last reset packet. Hence making sure that all the jobs were executed during each client-server session.

Third point is regarding 'reset' packets. There was no clear distinction between `RESET` packets sent in the beginning and end of a test run. Both reset packets performed the same set of tasks regardless of their order of arrival. This was addressed by clearly differentiating the two packets. The first reset job has three main tasks, first initialize the HashMap containing steady state probabilities to zero. Then wait for the first job to arrive and then log the frequency statistics provided by cpufreq-stats driver in Linux kernel. The last reset job marks the completion of all the jobs received by the processor. At first it logs the updated frequency statistics, the next task is to calculate the performance metrics such as mean service rate, mean response time and steady state probabilities and record them into excel sheet. And at last all the metric variables are set to zero.

Fourth point is related to the use of arrival queue. On arrival at the server, the job packets were first stored in an arrival queue and then passed one by one to the worker thread pool for execution. This technique creates an extra unwanted delay of transferring packets from arrival queue to the queue of worker thread pool. To avoid the delay, the arrival queue has been completely eliminated and the jobs are now directly added to the workQueue of ThreadPoolExecutor.

The last difference is that the old implementation used fixed size thread pool. This is inefficient because worker threads are idling during low workload. Plus it does not allow access to its own queue, hence giving rise to the problem above. To solve the issue, we are using a more adjustable constructor of ThreadPoolExecutor. This constructor provides with parameters such as maximumPoolSize, keepAliveTime etc. These parameters provide control to the user regarding the count of threads that should spawned and the lifetime of each threads. Now the threads are generated as and when needed.

Apart from the above improvements, a new feature has been added. A methodology to measure the steady state probability of the DTMC mentioned in Figure 4.2 has been successfully implemented. During each client-server session the system can achieve any number of states depending on the *arrival rate* of jobs. Each state represents the number of jobs in a system at a given point. The number of jobs in the system is the sum of number of jobs in the queue plus the job inside the server. In simple terms, the state probability means, the probability that the system was in a given state during a particular session. To measure the steady state probability of a system at a given *arrival rate*, the following methodology is adopted.

We know that only one job is serviced at any given point, therefore we can say that the state of the system changes only at two events namely, when a job arrives and when a job departs. We record the states with the help of a hashmap. The key of hashmap represents all possible states reached by the `Server` and the corresponding values represent the number of times that state occurred. The hashmap is updated right after the arrival of a job and after the job finished processing. The state was calculated at the occurrence of the above mentioned events and the corresponding value was incremented by one. In the end, the probabilities are calculated by dividing the frequency of each state with the total number of states.

To confirm the correctness of M/M/1 software mentioned in Section 3.1, we compared the steady state probabilities measured by the software to the ones obtained theoretically from the DTMC. The section below provides the comparison study.

3.3. Validation of M/M/1 Software

To prove the correctness of the implementation of our software, we adopted the following methodology: The steady state probabilities we obtained from our software

were compared with the steady state probabilities obtained from the DTMC of CPU enabled with ondemand governor as mentioned in Section 4.2. The DTMC was implemented using SHARPE tool. This tool implements algorithm to analyze markov models and provides with built-in function $prob(system\_name, state\_name)$ to calculate the steady state probability for a given system and state name. The arrival rate, service rate, sampling rate and utilization threshold were same for both M/M/1 software and SHARPE implementation. The only difference was in the transition probabilities. Our software does not need any transition probabilities to calculate the steady state probabilities, refer Section 3.2 for detailed explanation on how steady state probabilities are captured. On the other hand, the transition probabilities must be provided to the SHARPE model, we used the probabilities suggested in Section 4.2.1. The results obtained from the SHARPE model are completely theoretical while the results obtained from M/M/1 software are obtained from a real processor. Thus we can say that if the probabilities obtained from our software match with the probabilities obtained theoretically, the M/M/1 software is an accurate representation of M/M/1-FCFC queuing system.

Tables 3.1, 3.2 and 3.3 show the comparison between the steady state probabilities obtained with *arrival rates* (λ) 1, 10 and 20 jobs/second. Higher lambda means higher workload, which leads to a higher number of achieved states by the server. On analyzing the results, we noticed that mean absolute error for different values of λ is very low, thus indicating the correctness of our software.

Table 3.1.: Steady State Probability for λ=1, Mean Absolute Error= 0.131%

State	M/M/1 Software	SHARPE
0	95.606	95.643
1	4.068	4.227
2	0.325	0.130

Table 3.2.: Steady State Probability for λ=10, Mean Absolute Error= 1.189%

State	M/M/1 Software	SHARPE
0	66.350	62.354
1	22.919	26.305
2	7.440	7.273
3	2.588	2.638
4	0.617	0.957
5	0.069	0.347
6	0.017	0.126

Table 3.3.: Steady State Probability for λ=20, Mean Absolute Error = 0.548%

State	M/M/1 Software	SHARPE
0	32.171	29.540
1	21.687	25.732
2	14.633	13.585
3	10.098	9.477
4	6.794	6.612
5	4.518	4.613
6	3.397	3.218
7	2.191	2.245
8	1.529	1.566
9	1.036	1.093
10	0.747	0.762
11	0.527	0.532
12	0.314	0.371
13	0.187	0.259
14	0.102	0.181
15	0.042	0.126
16	0.025	0.088

4. Modelling Ondemand Governor

This chapter contains different markov chain models that we have considered in our research. Each of the models provide with solutions to generate power consumption and performance metrics for DVFS enabled processors.

4.1. CTMC representation of M/M/1-FCFS

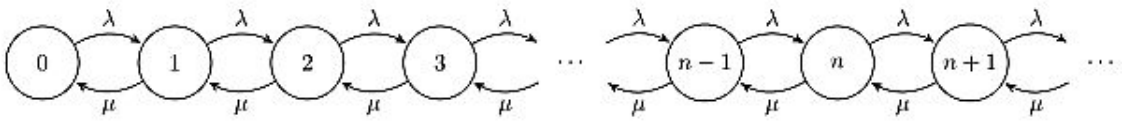

Figure 4.1.: M/M/1-FCFS Queuing Model

This continuous time markov chain CTMC is the simplest way of representing M/M/1-FCFS queuing system. Refer to Section 2.3 for basic introduction to M/M/1-FCFS. In this section we will discuss how to calculate various performance metrics and power consumption for this system. The first parameter to be calculated are the steady-state probabilities. According to [GBT06], a closed form solution is available to calculate the steady state probability of each state,

$$\pi_0 = 1 - \frac{\lambda}{\mu} \tag{4.1}$$

$$\pi_i = \pi_0 * (\frac{\lambda}{\mu})^i \tag{4.2}$$

$$P = (1 - \frac{\lambda}{\mu}) * P_{idle} + \frac{\lambda}{\mu} * P_{loaded} \tag{4.3}$$

P_{idle} and P_{loaded} can be easily measured with the help of the power meter. P_{idle} represents the power consumption of the processor when all the cores are 0 percent utilized. Whereas, P_{loaded} represents the power consumption when all cores are 100% utilized. During measurement of both the parameters the governor must be set accordingly. We set the governor to ondemand while measuring P_{idle} and P_{loaded} because we were using them to model the power consumption of ondemand governor.

4.2. DTMC model of Ondemand Governor

The discrete time markov chain DTMC in Figure 4.2 represents the detailed functioning of linux kernel version 2.6.9 of ondemand governor. Refer to Section 2.2 for details. Each state i_x represents the number of jobs in the system at a particular frequency, where i (i=0,1,2...) is the number of jobs and x ($0 < x < n$) represents frequency. At the end of each sampling interval δ the governor reads the utilization level of the processor and compares it to the threshold defined by the user. If the utilization exceeds the threshold, the frequency is switched from f_x to the highest possible frequency f_0. If the utilization remains below the threshold, frequency is reduced to one frequency (f_{x-1}) lower than the current frequency (f_x). The model assumes that the service time of the smallest job is greater than δ. This assumption assures that only one job is being served during a sampling interval. If there is a job waiting in the arrival queue, the processor remains at the highest frequency. The frequency switching takes place only when there is 1 or 0 job in the system. These states are represented by 1_x and 0_x. The next section describes how the state transition probabilities of this DTMC are calculated. To calculate mean response time T, we first calculate the mean number of jobs K, where π_i is the probability that there are i jobs in the system

$$K = \sum_{i=1}^{\infty} k * \pi_i \tag{4.4}$$

Then by applying Little's theorem [Lit], we can calculate T,

$$T = \frac{K}{\lambda} \tag{4.5}$$

To calculate the power consumption P, we need frequency specific power consumption values for idle and loaded CPU. Let the f_0,...,f_n be the available frequencies, where $P^0_{idle}, ..., P^n_{idle}$ represent the idle power consumption and $P^0_{loaded}, ..., P^n_{loaded}$ represent the power consumption when the CPU is 100% loaded. The formula for power consumption is as follows,

$$\begin{aligned} P = & \pi_{0_0} * P^0_{idle} + ... + \pi_{0_n} * P^n_{idle} \\ & + \pi_{1_0} * P^0_{loaded} + ... + \pi_{1_n} * P^n_{loaded} \\ & + (1 - \pi_0 - \pi_1) * P^0_{loaded} \end{aligned} \tag{4.6}$$

The utilization level **U** of a processor is calculated using the following formula,

$$U = 1 - \pi_0 \tag{4.7}$$

4.2.1. State Transition Probability

We know that the arrival of jobs occur according to Poisson process and the service times are exponentially distributed in M/M/1 queuing system. The inter-arrival

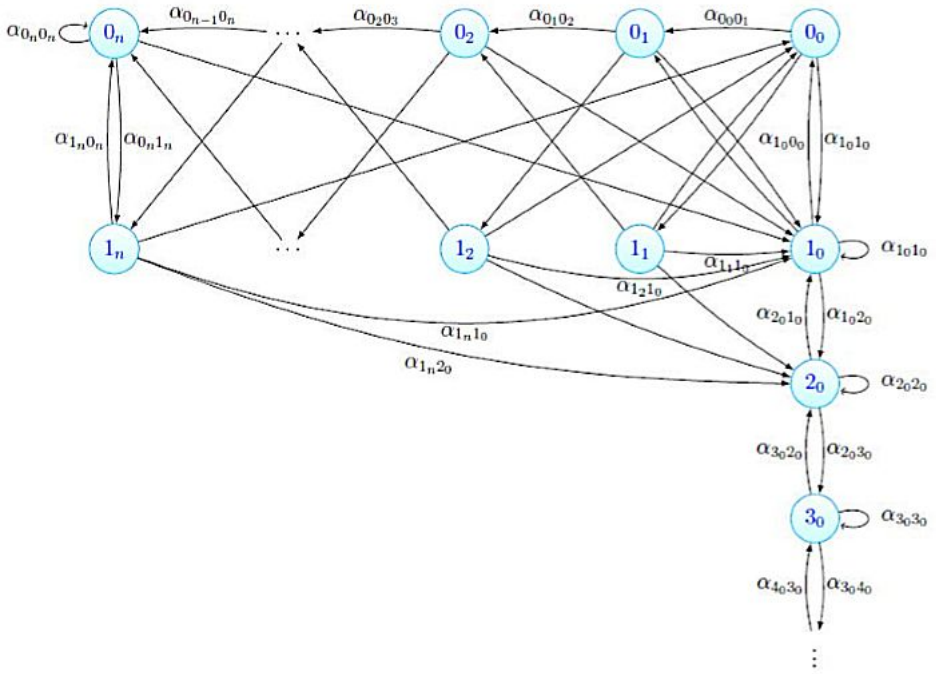

Figure 4.2.: DTMC of ondemand governor

times are exponentially distributed with parameter λ. According to Section 3.3.1 of Book [DB92], if t_n denotes the time of the n^{th} arrival with arrival rate λ. The intervals τ_n= t_{n+1} - t_n have the following probability distribution:

$$P\{\tau_n \leq \delta\} = 1 - e^{-\lambda\delta} \tag{4.8}$$

and if s_n is the service time of n_{th} job arrival, where service rate is μ

$$P\{\mathrm{s}_n \leq \delta\} = 1 - e^{-\mu\delta} \tag{4.9}$$

where δ is the duration of each interval. We will now discuss the transition probabilities between various states of the DTMC represented in Figure 4.2. I_k represents the current sampling interval, therefore I_{k-1} and I_{k+1} represent the previous and the next interval respectively. A_k and L_k stand for the number of jobs that arrived at and left the system in I_k. ρ_k represents the utilization of the processor in I_{k-1}, based on ρ_k frequency of I_k is decided. I_k is of length δ. The frequency x ranges from 0 to n, where 0 represents highest frequency and n is the lowest frequency. Each frequency corresponds to a service rate μ_x, ranging from μ_0 to μ_n. Below we discuss the probability of arrival and servicing of a job withing each sampling interval δ.

- Probability that no job arrives in interval δ is $e^{-\lambda\delta}$.
- Probability that a job arrives in interval δ is $1 - e^{-\lambda\delta}$.
- Probability that no job is serviced in interval δ is $e^{-\mu\delta}$.

- Probability that a job is serviced in interval δ is $1 - e^{-\mu\delta}$.

The equations below are the state transition probabilities α for DTMC shown in Figure 4.2

- There was no job in the system and no job arrived while the frequency remains at lowest.

$$\alpha_{0_n 0_n} \approx P[A_k = 0 \wedge \rho_k \leq u] = e^{-\lambda\delta} \tag{4.10}$$

- There was no job in the system and no job arrived while the current frequency was switched to one lower frequency.

$$\alpha_{0_x 0_{x+1}} \approx P[A_k = 0 \wedge \rho_k \leq u] = e^{-\lambda\delta} \tag{4.11}$$

- A job arrives in I_k, such that the utilization remains below threshold. The frequency remains the lowest.

$$\alpha_{0_n 1_n} \approx P[A_k = 1 \wedge L_k = 0 \wedge \rho_k \leq u] = e^{-\lambda(1-u)\delta}(1 - e^{-\lambda u\delta}) \tag{4.12}$$

- A job arrives in I_k causing the utilization to go above threshold. The frequency is switched to the highest.

$$\alpha_{0_x 1_0} \approx P[A_k = 1 \wedge L_k = 0 \wedge \rho_k > u] = e^{-\lambda u\delta}(1 - e^{-\lambda(1-u)\delta}) \tag{4.13}$$

- A job arrives in I_k, such that the utilization remains below threshold. The frequency is switched to one step lower.

$$\alpha_{0_x 1_{x+1}} \approx P[A_k = 1 \wedge L_k = 0 \wedge \rho_k \leq u] = e^{-\lambda(1-u)\delta}(1 - e^{-\lambda u\delta}) \tag{4.14}$$

- A job is serviced during I_k, but the utilization remains below threshold. The frequency remains at the lowest.

$$\alpha_{1_n 0_n} \approx P[A_k = 0 \wedge L_k = 1 \wedge \rho_k \leq u] = e^{-\lambda\delta}(1 - e^{-\mu_n u\delta}) \tag{4.15}$$

- A job is serviced in I_k, such that the utilization remains below threshold. The frequency is switched to one step lower.

$$\alpha_{1_x 0_{x+1}} \approx P[A_k = 0 \wedge L_k = 1 \wedge \rho_k \leq u] = e^{-\lambda\delta}(1 - e^{-\mu_x u\delta}) \tag{4.16}$$

- A job is serviced in I_k causing the utilization to go above threshold. The frequency is switched to the highest frequency

$$\alpha_{1_x 0_0} \approx P[A_k = 0 \wedge L_k = 1 \wedge \rho_k > u] = e^{-\lambda\delta}((1 - e^{-\mu_x\delta}) - (1 - e^{-\mu_x u\delta})) \tag{4.17}$$

- There is a job in processing causing the utilization to go above threshold. No job arrived or left. The frequency was switched to the highest frequency.

$$\alpha_{1_x 1_0} \approx P[A_k = 0 \wedge L_k = 0 \wedge \rho_k > u] = e^{-\lambda\delta} e^{-\mu_x \delta} \tag{4.18}$$

- A job is already in processing, when another job arrives it causes the utilization to go above threshold. The frequency is switched to highest.

$$\alpha_{1_x 2_0} \approx P[A_k = 1 \wedge L_k = 0 \wedge \rho_k > u] = (1 - e^{-\lambda\delta}) e^{-\mu_x \delta} \tag{4.19}$$

for $i > 0$:

- There are already jobs in the system causing the utilization to be above threshold. No job arrives and departs from the system. The frequency remains at the highest.

$$\alpha_{i_0 i_0} \approx P[A_k = 0 \wedge L_k = 0 \wedge \rho_k > u] = e^{-\lambda\delta} e^{-\mu_0 \delta} \tag{4.20}$$

- There are already jobs in the system causing the utilization to be above threshold. A job arrives into the system. The frequency remains at the highest.

$$\alpha_{i_0 (i+1)_0} \approx P[A_k = 1 \wedge L_k = 0 \wedge \rho_k > u] = 1 - e^{-\lambda\delta} \tag{4.21}$$

- There are already jobs in the system causing the utilization to be above threshold. A job is serviced and leaves the system. The frequency remains at the highest.

$$\alpha_{(i+1)_0 i_0} \approx P[A_k = 0 \wedge L_k = 1 \wedge \rho_k > u] = 1 - e^{-\mu_0 \delta} \tag{4.22}$$

With the help of DTMC mentioned above, steady state probabilities can be easily calculated using SHARPE tool, which is shown in next section.

4.3. DTMC implemented in SHARPE

SHARPE (Symbolic Hierarchical Automated Reliability and Performance Evaluator) is a modelling tool that analyzes stochastic models such as M/M/1 queuing system. It includes algorithms to analyze a Markov chain and generate steady state probabilities, mean time of system failure, mean time of system restoration etc. It provides a coding language and supports various built-in functions. The user can customize the code to calculate performance metrics such as mean response time and power consumption. The built-in function that was used to implement the DTMC mentioned in Figure 4.2 is `prob(system_name, state_name)`. It returns the steady state probability for the mentioned state name. Lets look at the standard code structure:

- At first we define all the constants utilization threshold, sampling interval, arrival rate, service rate and power consumption. For example

```
bind
lambda 10
delta 0.01
end
```

this statement assigns 0.01 seconds to delta (sampling interval) and 10 jobs/second to lambda (arrival rate).

- Then comes the state transition probabilities between all states (1_2 1_0) of the DTMC, for example

```
markov state6
1_2 1_0 (e^(-1*lambda*delta))*(e^(-1*mu2*delta))
end
```

this statement inserts the transition probability from state 5_0 to 4_0 where δ, μ_0 have already been defined with other constants.

- The last section is for defining the functions for calculating steady state probability and performance metrics such as mean response time, power consumption and utilization for the given arrival rate, sampling interval and utilization threshold.

```
var k \
pi_1_sum+(2*prob(state_6 ,2_0))+(3*prob(state_6 ,3_0))+
(4*prob(state_6 ,4_0))+(5*prob(state_6 ,5_0))+
(6*prob(state_6 ,6_0))


var MRT \
(k/lambda)*1000

expr MRT
```

The above code is the example of how mean response time can be calculated using equation 4.4 and 4.5. Similarly steady state probabilities(from π_0 to π_n), power consumption (4.6) and utilization (4.7) are calculated using there respective formulas.

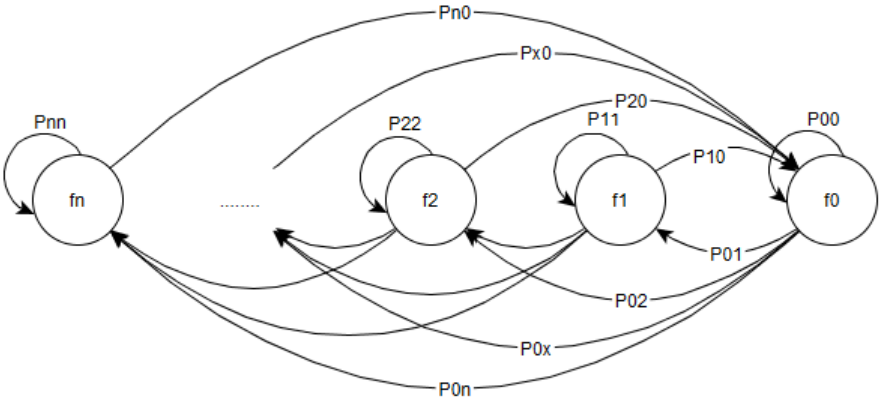

Figure 4.3.: Proposed DTMC of ondemand governor-NewDTMC

4.4. Empirical Modelling-Proposed DTMC for ondemand governor

The DTMC represented in Figure 4.3 is based on the new version 2.6.16 of ondemand governor. Each state x represents a frequency supported by the processor from f_0,.., fn, such that $f_0 < x < fn$. Each edge represents a switching probability P_{ij} from between two frequencies. This model can be classified as a Markov process, since the future states are only dependent on the present state and not the past states. The state transition matrix can been obtained from using the data provided by CPUFreq statistics *"/sys/devices/system/cpu/cpux/cpufreq/stats"*:

time_in_state $T(f_i)$: Amount of time spent in a frequency (GHz). The unit of $T(f_i)$ is based on the common sampling interval (δ) of cpufreq's ondemand governor, which in our case is 10 ms. Table 4.1 demonstrates an example of time_in_state table.

Table 4.1.: time_in_state

f_i	time
2.6	59638
2.3	124
1.8	110
1.3	153
0.8	262

trans_table $T(f_i \Rightarrow f_j)$: It is a 2-dimensional matrix that represents the count of number of transitions from f_i to f_j. The first column represents $T(f_i$ and the first row represent $f_j)$. Table 4.2 demonstrates an example of time_in_state table. The value of self-transition $T(f_i \Rightarrow f_i)$ is not generated by Linux cpufreq-stats driver.

Table 4.2.: trans_table

f_i/f_j	2.6	2.3	1.8	1.3	0.8
2.6	-	50	60	82	75
2.3	49	-	1	1	0
1.8	62	1	-	2	0
1.3	83	0	2	-	1
0.8	73	0	2	1	-

In Section 4.4.1, we explain the methodology of calculating state transition matrix from the above data.

4.4.1. State Transition Matrix

As we know from Section 2.2.1, the ondemand governor checks the utilization of the system at fixed sampling intervals. Based on the utilization in the past interval, it decides whether to switch the frequency or not. This implies that number of sampling intervals are equal to the number of decisions taken by ondemand governor to switch the frequency of the processor. The second column of Table 4.1 represents the number of sampling intervals during which the processor was in respective frequency. Therefore, we can conclude that the time spent in each frequency is equal to the number of switching decisions taken by processor. We do not need explicitly calculate steady state probability. SHARPE tool automatically assigns self transition probabilities to all the states of the NewDTMC model.

Following the above conclusion, the conditional probability P_{ij} is shown in Equation 4.23

$$P_{ij} = T(f_i \Rightarrow f_j)/T(f_i) \tag{4.23}$$

The state transition probabilities are mentioned in the table 4.3 below:

Table 4.3.: trans_table with state transition probabilty

f_i/f_j	2.6	2.3	1.8	1.3	0.8
2.6	-	0.084	0.101	0.137	0.126
2.3	39.516	-	0.806	0.806	0.000
1.8	56.364	0.909	-	1.818	0.000
1.3	54.248	0.000	1.307	-	0.654
0.8	27.863	0.000	0.763	0.382	-

4.4.2. Calculation of Steady State Probabilities

To calculate the steady state probability, the new model mentioned in Figure 4.3 was implemented in SHARPE tool. The state transition probabilities obtained in Section 4.4.1 was given as the input to the model. And the output was steady state probabilities P_0, ..,P_n, which defines the probability that the system was in a given frequency.

Another method to calculate the steady state probability is to use the time_in_state table mentioned in Section 4.4. The sum $T = \sum_{i=0}^{n} T(f_i)$ of all the times spent in each frequency is the total time spent in the system. The formula to calculate steady state probability for each frequency is shown in Equation 4.24

$$P_i = T(f_i)/T \tag{4.24}$$

The steady state probabilities are used to calculate the service rate of the processor for a particular utilization, we suggest the following formula for service rate μ

$$\mu = \sum_{i=0}^{n} P_i * \mu_i \tag{4.25}$$

where μ_i is the service rate of the processor when the frequency is fixed to i.

4.4.3. Proposed methodology to predict the steady state probabilities of NewDTMC

In this section we derive a model to generate steady state probabilities of NewDTMC. Based on the frequency statistics obtained from each of the servers, we analyzed the behavior of ondemand governor. The Figures 4.4 and 4.5 shows the steady state probability of each frequency with varying utilization levels for AMD and INTEL processors respectively. For AMD processor we have considered all five of its frequencies, whereas for INTEL server every third frequency was chosen such that the frequencies are logically comparable to frequencies of AMD server. The graphs show that the ondemand governor exhibits similar behavior on different architectures. A best fit solution was generated for each of the graphs in Figures 4.4 and 4.5. Eureqa, an artificial intelligence power modelling engine was used to generate these solutions.

Table 4.4.: Steady state equations for AMD server

Frequency	Best fit solution
2.6 GHz	$1.166 * U + 64.73 - 0.008 * U^2$
2.3 GHz	$0.025 * U - 0.106 - 0.0002 * U^2$
1.8 GHz	$0.033 * U - 0.025 - 0.0003 * U^2$
1.3 GHz	$0.000 * U + 0.966 - 8.661e - 5 * U^2$
0.8 GHz	$0.007 * U + 136.834/U - 1.620$

Table 4.5.: Steady state equations for INTEL server

Frequency	Best fit solution
2.6 GHz	$1.189 * U + 60.929 - 0.008 * U^2$
2.3 GHz	$0.011 * U - 0.039 - 0.0001 * U^2$
2.0 GHz	$0.008 * U - 0.005 - 8.495e - 5 * U^2$
1.7 GHz	$0.002 * U + 0.170 + 3.921e - 5 * U^2$
1.4 GHz	$0.005 * U + 0.356 + 9.677e - 5 * U^2$
1.2 GHz	$0.021 * U + 132.314/U - 3.381$

Tables 4.4 and 4.5 show that steady state probability is a function of utilization U. Although the shape of the graph is similar for comparable frequencies for two different architectures, we were unable to formulate a universal equation that could generate the steady state probabilities of ondemand governor independent of the architecture for a given markovian load.

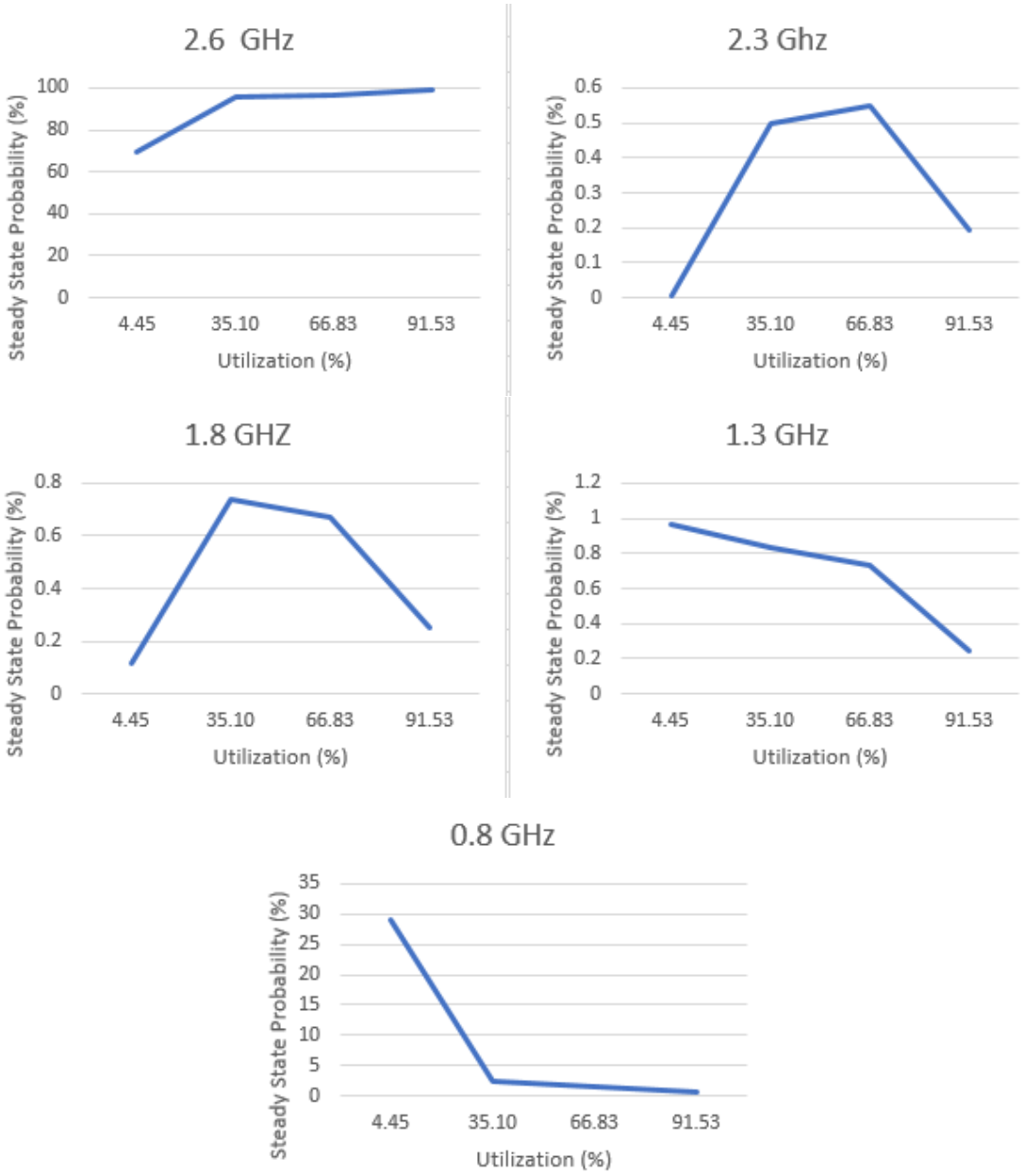

Figure 4.4.: Steady State probability of NewDTMC-AMD

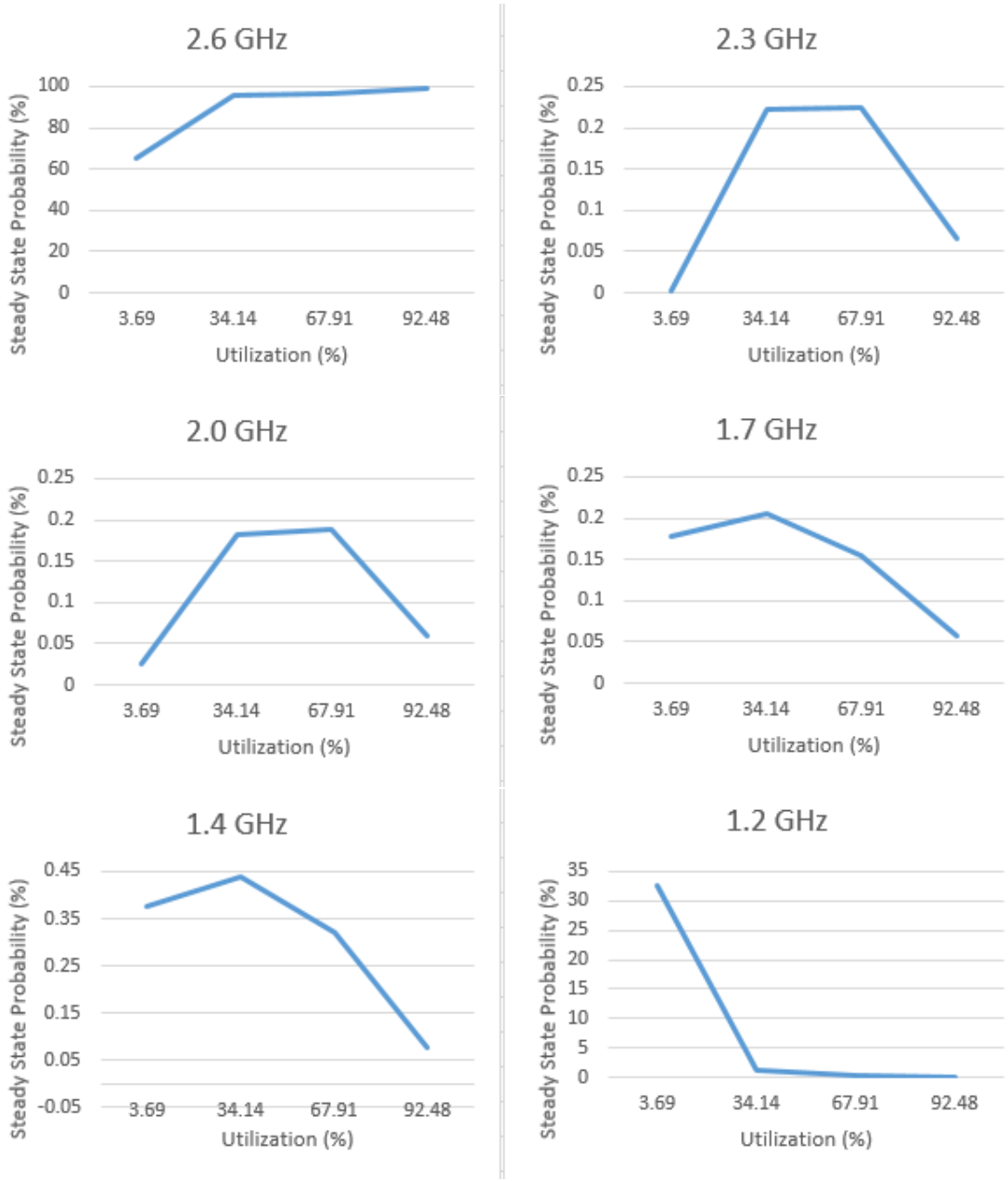

Figure 4.5.: Steady State probability of NewDTMC-INTEL

5. Evaluation

This chapter explains the evaluation setup and various comparison studies that are performed using *M/M/1 software*. The markov chain models mentioned in Chapter 4 are compared against the measurements obtained from *M/M/1 software*. Their accuracy in modelling power consumption and performance metrics for a processor enabled with ondemand governor is calculated.

5.1. Client-Server Setup

The components of M/M/1 software are illustrated in Section 3.1. In this section we provide brief description of hardware used during evaluation and also discuss the role of each component.

1. Host h: Two different architectures have been used to perform the analysis:

   - INTEL: The server is a Dell PowerEdge R720 rack server. It contains Intel Xeon(R) CPU E5-2650 v2 @ 2.60GHz processor. It has 2 sockets, 8 cores per socket and therefore in total 16 cores. The processor was released in Q3'13. Hyper-threading has been disabled in the processor, each physical processor is assigned exactly one thread. Possible CPU frequencies are: $f_{14} = 1.2$ GHz, $f_{13} = 1.3$ GHz, $f_{12} = 1.4$ GHz, $f_{11} = 1.5$ GHz, $f_{10} = 1.6$ GHz, $f_9 = 1.7$ GHz, $f_8 = 1.8$GHz, $f_7 = 1.9$ GHz, $f_6 = 2.0$ GHz, $f_5 = 2.1$ GHz, $f_4 = 2.2$ GHz, $f_3 = 2.3$ GHz, $f_2 = 2.4$ GHz, $f_1 = 2.5$ GHz, $f_0 = 2.6$ GHz. Its maximum turbo frequency is 3.40 GHz. For the purpose of analysis turbo boost has been disabled. Ubuntu 16.04.1 LTS is installed as the operating system. CPU frequency scaling is implemented in the Linux kernel via the infrastructure known as **cpufreq**. We use the ondemand governor version 2.6.9 provided by cpufreq for our tests.

   - AMD: This server is equipped with AMD Opteron(tm) 4180 Processor. It has 2 sockets, 6 cores per socket and therefore in total 12 cores. This processor was introduced in June 2010. It does not support hyper-threading and turbo frequency boost. Possible CPU frequencies are: $f_4 = 0.8$ GHz, $f_3 = 1.3$ GHz, $f_2 = 1.8$ GHz, $f_1 = 2.3$ GHz, $f_0 = 2.6$ GHz. Similar to the Intel server, the operating system is Ubuntu 16.04.1 LTS and the governor used is ondemand.

2. Server s: Server s runs on the host. It receives jobs from the load generator and passes it to the processor for execution in a first come first serve fashion. Each job contains a random integer n. The random numbers are exponentially distributed. n determines the service time of the job at h, as a result the service time is exponentially distributed with parameter $1/\mu$. All the jobs are pinned to a different core that server itself. This assures that, no other tasks are executed along with the jobs.

3. Client c: The client runs on a virtual machine with an Ubuntu as an operating system. The hypervisor is an Oracle Virtual Box. The VM host is an HP Pavilion notebook equipped with Intel(R) Core(TM) i3-4030U CPU @ 1.90GHz processor with a Windows 10 operating system. The client is responsible for generating jobs for server. Jobs is sent using UDP. Each job contains an exponentially distributed number n (size of packet). The jobs are generated such that the inter-arrival times are exponentially distributed with rate λ. The client sends reset packets at the start and end of each session.
   Even though the UDP protocol is unreliable, no packets were lost during our experiments. We calculated and compared the number of packets sent by the client to the number of packets received by the server. No loss of packets was observed.

4. Power meter: Server s is attached to the ZES LMG 500 power meter. It measures the power consumption of the host 2 times per second. The power consumption of host h when s is servicing jobs, is measured directly from the power meter.

5.1.1. Selection of input parameters

The client requires three input parameters: Arrival rate, duration and repeat value. Choosing suitable values of these parameters is critical to obtain correct results.

- Arrival Rate (λ): The arrival rate of jobs is directly proportional to the amount of utilization (U) it causes at the server. Therefore it is be chosen in a way that it is equally distributed between 0% to 100% of utilization. Table 5.1 and 5.2 show the values that were chosen to perform the evaluation tests.

Table 5.1.: Arrival rate and utilization for AMD server

Input λ	Obtained λ	U
1	0.97	4.45
8	8.00	35.10
16	15.59	66.83
22	21.41	91.53

Table 5.2.: Arrival rate and utilization for INTEL server

Input λ	Obtained λ	U
1	1.01	3.69
10	9.84	34.14
20	19.57	67.91
28	26.90	92.48

- Duration: The duration of each test run is chosen such that the λ given as the input the client is as close as possible to the measured arrival rate at the server. After trying different durations of time, 10 minutes was found to be the most suitable and practical. Each test run has a duration of 10 minutes and is repeated 3 times. All the presented results are an average of 3 test runs.
- Repeat: The repeat value is set to 50,000 for both the servers to obtain the mentioned levels of utilization. The repeat value was carefully chosen such that the the measured service rate satisfied the formula of utilization for CTMC i.e.

$$Utilization = \frac{ArrivalRate}{ServiceRate} \tag{5.1}$$

5.1.2. Other parameters

These measurements are needed for calculating performance and power metrics from CTMC and DTMC of ondemand governor.

- P_{loaded} and P_{idle}: P_{loaded} is the measured power consumption when the processor is 100 percent utilized and P_idle is power consumption of an idle processor, while the governor is set to ondemand. To measure frequency specific P_{loaded_i} and P_{idle_i}, where i represents the specific frequency, the governor was set to userspace and frequency was fixed to i.
- μ_i: It stands for the mean service rate (MSR) generated by the processor at a given frequency i. M/M/1 software was used to calculate the MSR. Five measurements were recorded for each frequency and their average was calculated for the final value of μ_i.

5.2. Comparison between CTMC and measured values

Below is the comparison study of results obtained from CTMC mentioned in section 4.1 and the implementation of M/M/1 software described in section 3.1. Mean

response time and power utilization's are presented for different levels of utilization of the processors. The results from both processors, Intel and AMD, show that the simple CTMC of M/M/1 is suited to model performance and power of DVFS enabled processors.

At first we discuss the mean response time. The CTMC models shows an average relative error of 5.06% for INTEL server as compared to 7.50% for AMD server. The error is gradually increasing with increasing workload for both servers. The estimation of power consumption by the CTMC model is highly accurate with a negligible average relative error of 0.30% for INTEL server and 2.38% for AMD. The draw back of this model is that it does not consider the sampling interval and utilization threshold in its formulas, thus leaving no scope for optimization based on these two parameters.

Table 5.3.: CTMC vs M/M/1 software-INTEL

Mes. U	Mod. MRT	Mes. MRT	Error	Mod. Power	Mes. Power	Error
3.84	38.47	38.78	0.80	106.71	106.74	0.03
34.14	52.72	50.78	3.82	110.04	109.75	0.26
67.91	107.14	103.94	3.08	113.73	113.32	0.36
92.48	464.28	412.90	12.44	116.46	115.97	0.42

Table 5.4.: CTMC vs M/M/1 software-AMD

Mes. U	Mod. MRT	Mes. MRT	Error	Mod. Power	Mes. Power	Error
4.45	48.01	47.98	0.06	79.30	82.76	4.18
35.10	67.45	65.05	3.69	92.82	96.27	3.58
66.83	129.69	120.45	7.67	106.89	108.56	1.54
91.53	524.35	442.26	18.56	117.87	118.14	0.23

5.3. Comparison between DTMC and M/M/1 software

In this section, we will discuss the accuracy of DTMC model for utilization based ondemand governor as mentioned in section 4.2. The DTMC is implemented in SHARPE. This tool generates the steady state probabilities (π_i) where i stands for the states of the DTMC, as represented in Figure 4.2.

The steady state probabilities are used to calculate power consumption and mean response time of the processor. Tables 5.5 and 5.6 show the comparison between the results obtained from the DTMC model against the values obtained from M/M/1

software. From the aforementioned results we conclude that the DTMC model is a good representation of a processor running with ondemand governor. The mean response time was estimated with 10.4% average relative error for INTEL server and 7.59% for AMD. While the error in estimating power consumption is 0.53% for INTEL and 3.51% for AMD server.

The modelled results are a good estimation of the power consumption and MRT, in spite of the differences between the version ondemand governor considered in the DTMC model and the version of ondemand governor running on the processor while the measurements were taken, details can be found in Section 2.2.1. Therefore, we conclude that the changes made to the ondemand governor seem to have no noticeable effect on the power consumption and MRT of the processor.

Table 5.5.: DTMC vs M/M/1 software-INTEL

Mes. U	Mod. MRT	Mes. MRT	Error	Mod. Power	Mes. Power	Error
3.84	43.22	38.78	11.45	107.13	106.74	0.37
34.14	56.65	50.78	11.56	110.68	109.75	0.85
67.91	109.17	103.94	5.03	114.08	113.32	0.67
92.48	464.28	412.90	12.44	116.39	115.97	0.36

Table 5.6.: DTMC vs M/M/1 software-AMD

Mes. U	Mod. MRT	Mes. MRT	Error	Mod. Power	Mes. Power	Error
4.45	52.43	47.98	9.27	78.66	82.76	4.95
35.10	71.06	65.05	9.24	96.42	96.27	0.16
66.83	129.87	120.45	7.82	112.84	108.56	3.94
91.53	424.37	442.26	4.05	124.04	118.14	4.99

5.4. Results obtained from NewDTMC as compared to M/M/1 Software

The new version of ondemand governor does not follow a simple set of rules like the older version, in the older version the frequency was reduced step by step when the utilization of the processor remained under the utilization threshold. On the other hand, the current version jumps directly to the lowest frequency that would keep the processor 80% utilized, whenever it noticed a utilization lower than 20%. To model this scenario, we must consider the possible transitions from states 0_i and 1_i of the DTMC mentioned in Figure 4.2 to each of its lower states and introduce a separate threshold parameter for frequency down-scaling. These changes increase the complexity of the Markov chain and make it complicated to implement. To

avoid this problem we created a simpler model. Its states represent the frequency of the processor instead of number of jobs in the system. The transition probabilities and steady state probabilities are calculated as mentioned in Section 4.4. With the help of steady state probabilities we calculate service rate of the processor running with ondemand governor, using Equation 4.25. Once the service rate is established with the help of the above model, we use the simple CTMC model of M/M/1-FCFS mentioned in Section 2.1 to calculate the mean response time and power consumption.

A comparison study between the results obtained using the above approach as compared to the results obtained from M/M/1 software is presented below. Tables 5.7 and 5.8 show the comparison between the mean service rate empirically obtained from the newly suggested DTMC in section 4.4 and the measurements obtained from M/M/1 software.

Table 5.7.: MSR of INTEL server for NewDTMC model

Mes. U	NewDTMC MSR	Mes. MSR	Error
3.84	24.03	27.03	11.10
34.14	28.97	28.80	0.59
67.91	29.19	28.93	0.90
92.48	29.41	29.05	1.24

Table 5.8.: MSR of AMD server for NewDTMC model

Mes. U	NewDTMC MSR	Mes. MSR	Error
4.45	19.22	21.82	11.92
35.10	23.42	22.83	2.58
66.83	23.62	23.31	1.33
91.53	23.85	23.34	2.19

The results show that "NewDTMC" model can predict the service rate with a mean relative error of 3.68% for INTEL and 4.5% for AMD. Based on MSR obtained from the DTMC, other metrics are calculated and compared with measurements of M/M/1 software in Tables 5.9 and 5.10. The mean relative error in estimating MRT is 4.9% for INTEL and 6.23% for AMD. The estimation of power consumption is even more accurate, 0.25% for INTEL and 2.68% for AMD.

Table 5.9.: MRT and Power of INTEL server for NewDTMC model

Mes. U	NewDTMC MRT	Mes. MRT	Error	NewDTMC Power	Mes. Power	Error
3.84	43.43	38.78	11.99	106.75	106.74	0.01
34.14	52.27	50.78	2.93	110.02	109.75	0.25
67.91	104.03	103.94	0.09	113.66	113.32	0.30
92.48	397.14	412.90	3.82	116.34	115.97	0.32

Table 5.10.: MRT and Power of AMD server for NewDTMC model

Mes. U	NewDTMC MRT	Mes. MRT	Error	NewDTMC Power	Mes. Power	Error
4.45	54.79	47.98	14.19	79.56	82.76	3.87
35.10	64.84	65.05	0.32	92.43	96.27	3.99
66.83	124.43	120.45	3.30	106.49	108.56	1.91
91.53	410.83	442.26	7.11	117.02	118.14	0.95

5.5. Behaviour of Ondemand Governor

Our **newDTMC** model depends on steady state probabilities of the available frequencies of a processor. It is interesting to analyze the behaviour of the ondemand governor and how it affects the steady state probabilities at a given workload. Figures 5.1 and 5.2 show that for exponentially distributed workload, the process spends most of the time at the highest frequency thus behaving more like a performance governor. Both the servers show similar behaviour for similar levels of utilization. It is suspected that this behaviour may be caused by Markovian arrival and service processes, which is causing the processor to jump to the highest frequency with the arrival of each job. This is suggested as future work, to confirm if such a behaviour is caused by other types of workloads as well.

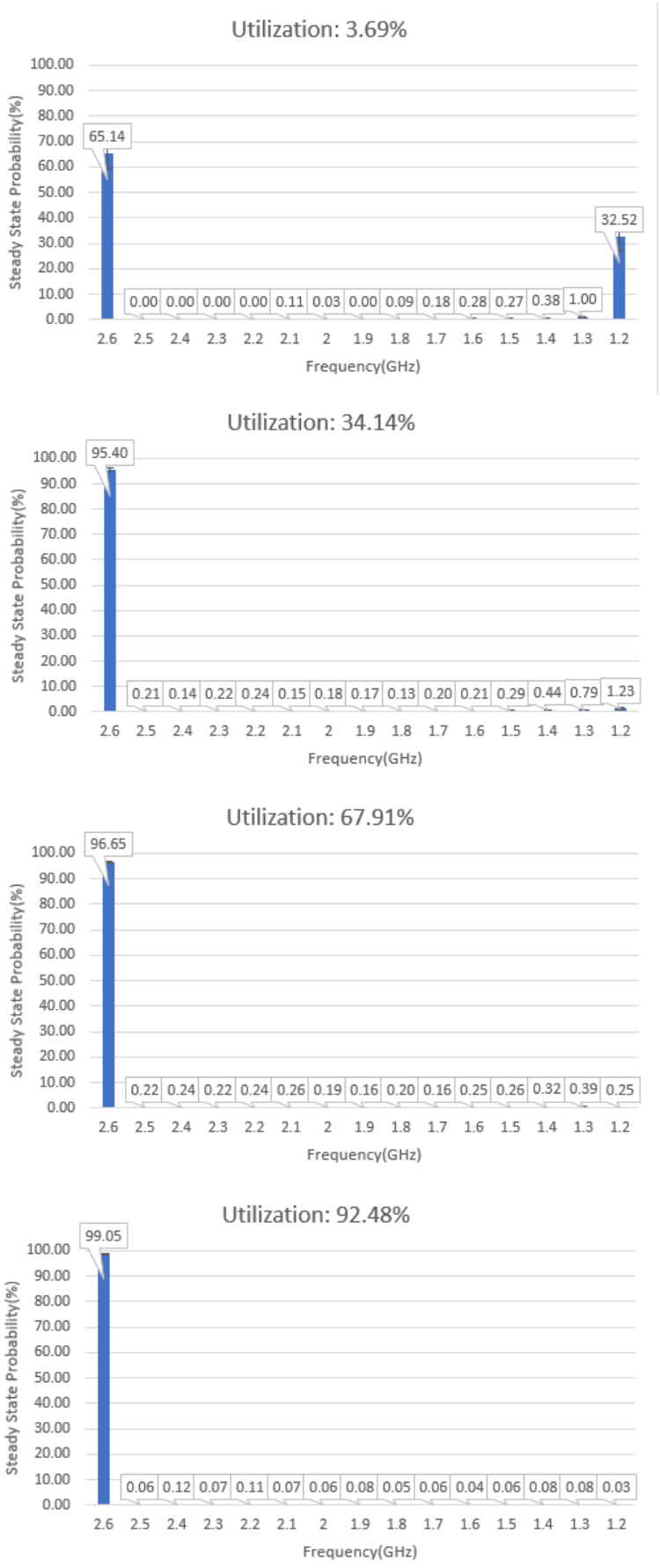

Figure 5.1.: Steady state probabilities of INTEL server

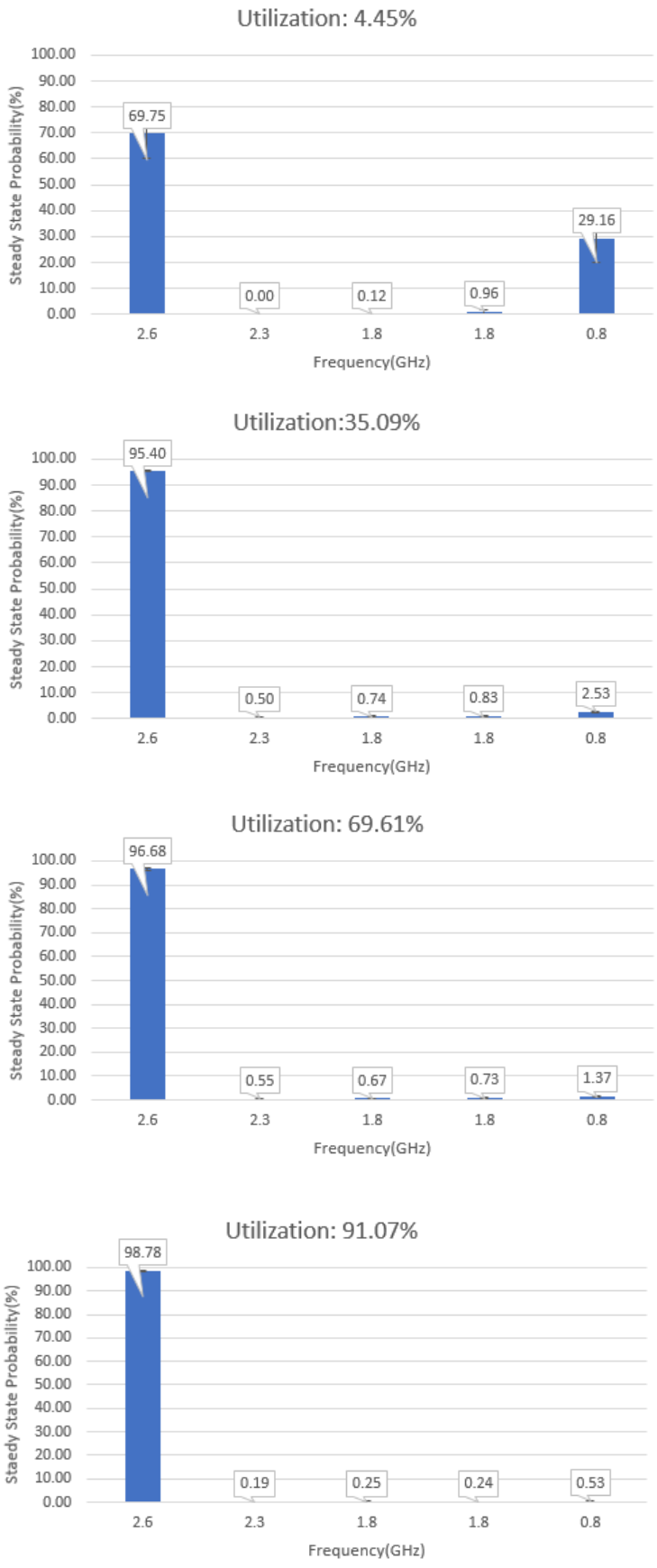

Figure 5.2.: Steady state probabilities of AMD server

6. Future Work

6.1. Optimization

The research paper [BNdM16] suggests that adjusting utilization threshold (u) and sampling rate (δ) according to the arrival rate leads to reduced power consumption while keeping the mean response time below the threshold. The above optimization is suggested based on the DTMC suggested in Section 4.2. The DTMC represents the old version 2.6.9 of ondemand governor, which is no longer available for use. We use the M/M/1 software mentioned in Section 3.1 to study the performance and power consumption of the INTEL processor equipped with latest version 2.6.16 ondemand governor.

The graphs mentioned in Figure 6.1 shows the behaviour of ondemand governor with varying threshold. It can be clearly noticed that the power consumption stays almost constant with change of utilization threshold, for all values of arrival rate. This concludes that utilization threshold does not play a significant role towards optimizing power consumption. The same is true for sampling interval with respect to power consumption, as shown in Figure 6.1. Next we will discuss the effect of varying u and δ on the mean response time of a processor. For arrival rates 1, 10 and 20 the graphs show almost no variation in MRT. However for arrival rate 28, which corresponds to approx 92 percent of overall server utilization, the power consumption and MRT show varying values. This behaviour presents with a possibility of optimizing u and δ for achieving lower mean response time during high workloads on processor.

The possible explanation for such behavior is as following. As seen in Section 5.5, the markovian workload causes the processor to jump between the highest and the lowest frequency, hence not allowing the governor to optimize. Another explanation could be the intelligent behaviour exhibited by the latest version of ondemand.[VP] The governor now intelligently jumps to the suitable frequency based of current workload. Thus eliminating the need for optimization. As future work, the behaviour of ondemand governor can be studied for different types of workload.

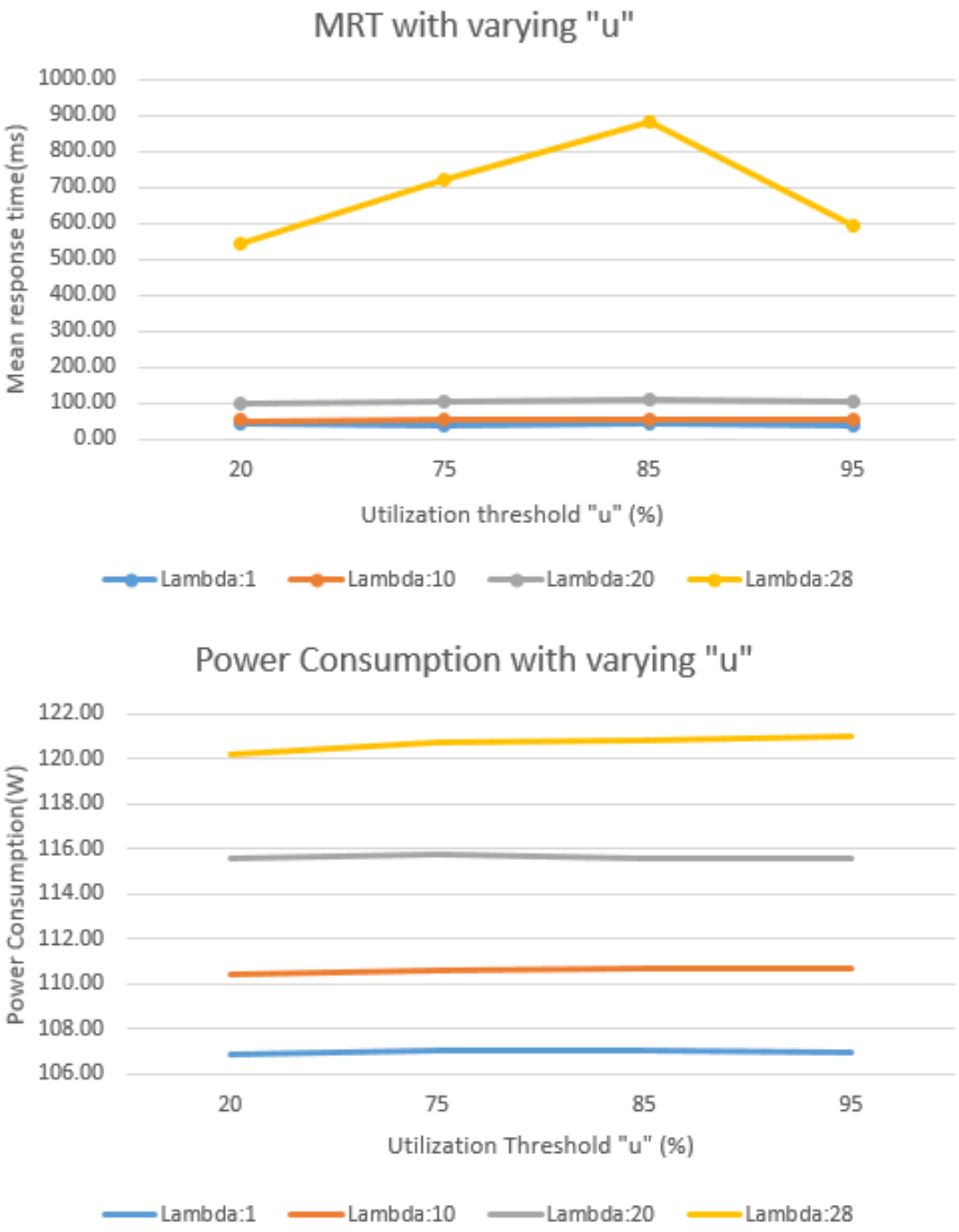

Figure 6.1.: Behaviour of ondemand governor with varying utilization threshold

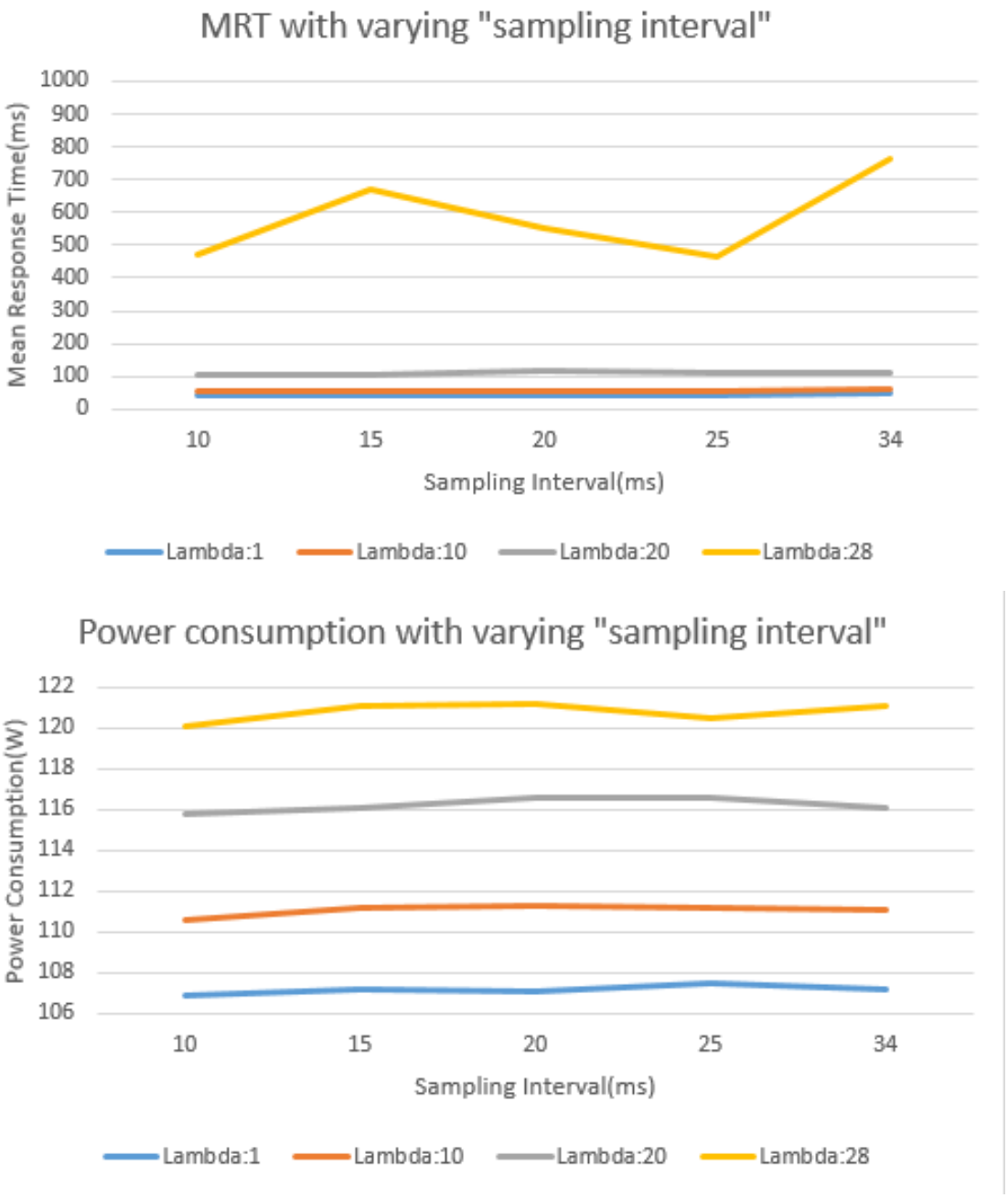

Figure 6.2.: Behaviour of ondemand governor with varying sampling interval

6.2. Multi-core processors

The M/M/1 software mentioned in Section 3.1 can be easily converted to work as a M/M/m queuing system by changing the value of parameter **corePoolSize** to m servers while initializing the ThreadPoolExecutor. The ThreadPoolExecutor will equally divide the load between the m servers. Thus by tweaking M/M/1 software, it can be used to capture the service time and mean response time for the given workload. Analyzing the behaviour of ondemand governor for multicore processors is suggested as future work.

6.3. Significance of the "size" of the work packet

During all the test runs performed on the M/M/1 software, the selection of repeat value, which determined the size of the work packet was crucial. The repeat value must generate a service rate that satisfies the following condition:

$$Utilization = ServiceRate/ArrivalRate \tag{6.1}$$

We obtained this value of service rate at a repeat value of 50,000. This repeat value worked fine for both INTEL and AMD server. This shows that satisfying the above formula is critical for obtaining correct results from the software. We suggest that more research can be done exploring the effect of various repeat values on the behaviour of M/M/1 software.

6.4. Behaviour of Ondemand governor for non-markovian workload

In this thesis, we considered Markovian arrival and service rate were for modelling the power and performance the ondemand governor. In Section 4.4.3, we show that the behaviour of ondemand governor is a function of utilization of the processor. The behaviour of non-markovian workloads according to the NewDTMC described in Section 4.4 is also a possible field to study. We recommend distributions such as *Degenerate*, *General* and *Phase-type* distributions for modelling arrival and service processes.

7. Related Work

This section presents a variety of research work done in the field of CPU frequency scaling and Markov modelling. The research paper [Kar13] suggests a model for service rate control system, this model improves the efficiency of CPU frequency control mechanism.

They have introduced a bi-objective processing performance index that considers cost of energy and service quality index corresponding to operating mode of the processor. A mapping has been implemented between optimal service rate level corresponding to the observed workload. This paper focuses on how power consumption can be reduced if optimal service rates are provided by the system. The results obtained confirms that power efficiency and Quality of Service (QoS) are conflicting objectives.

The second paper [JKL+13] suggests a power management scheme that control DVFS core shutdown strategies for application processors. Apart from considering CPU utilization as the parameter for frequency scaling, they suggest that the characteristics of the application must also be considered to achieve better power saving while guaranteeing QoS. Already existing Android power management schemes such as hotplug and Linux ondemand governor under-performs in terms of power saving compared to application aware power management scheme.

Another recent research paper [HPDL15], talks about a memory aware CPU-GPU DVFS for mobile games. In recent mobile phones, GPU comes integrated with multicore processors for graphic intensive applications. These application cause a very high power consumption. Hence the paper suggests an approach that combines CPU utilization with memory access footprint to perform frequency scaling. The suggested governor works with CPU, GPU and memory governor, and it takes a frequency switching decision each time a few frame is pushed out for display.

Next paper [gKCE+12] concentrates on energy efficient eDVFS for multicore processors, that is more efficient than the ondemand CPU governor. This approach considers energy consumption of both CPU and memory intensive workloads. The eDVFS considers memory traffic as a threshold to manipulate frequency. There is no consideration of QoS in decision making.

However the next paper [ADD12] takes the service level agreement into consideration. The SLA based governor receives periodic details on performance and expected SLA. The application communicates its performance periodically to the governor.

Based on application's SLA requirement, the governor makes frequency switching decisions. The application considered here is online transaction processing. The power consumption was found to be lower than that of the traditional ondemand governor.

The most recent paper [BKZ+15] was published in 2015, it combines Markov modelling with DVFS in multicore processors. It works by detecting phases in the workload subjected by an application. These phases have corresponding performance and power attributes. Once the phases have been detected, a static power schedule is constructed and frequency scaling is performed based on the schedule. This strategy works best for predictable and iterative workload.

All the above works show the wide variety of research that is being done to achieve an optimal solution for reducing power consumption while providing an acceptable quality of service.

8. Conclusion

In our work, we have developed an improved software implementation referred to as the M/M/1 software. This software replicates the behaviour of M/M/1 queuing system and calculates the power and performance metrics of the processor. The software also successfully captures the steady state probability of number of jobs in the system and the frequency statistics of the core on which jobs are processed. With the help of the metrics obtained from this software we can confirm the accuracy of various Markov chain models.

In this thesis, we have studied the accuracy of two of the previously suggested models: CTMC of M/M/1-FCFS queuing system and DTMC of CPU-bound service running on a DVFS-enabled processor with ondemand governor. Both models show promising results when compared to the software measured values, but we discovered that the DTMC of ondemand governor is outdated and represents the older version of the ondemand governor. Therefore we presented a new approach to model the current version of ondemand governor. This approach considers each of the available frequency f_i of the processor as a state and the steady state probability for each frequency is empirically obtained from the frequency statistics. The results obtained from the newDTMC were compared to the measured metrics and the model was found to be suitable.

We performed tests to study the criteria for optimizing the the ondemand governor. The sampling rate and utilization threshold do not play a significant role in reducing power consumption and mean response time for the current workload. Although these test were preliminary and require further research for more conclusive results.

A. Appendix

Listing A.1: Sample SHARPE code for DTMC of ondemand governor for arrival rate of 4.8 jobs/sec

```
format 8
factor on

bind
u    .95
e    2.71828
delta    .01
lambda 4.811
mu14    13.62
mu13    14.72
mu12    15.87
mu11    16.81
mu10    17.82
mu9    19.28
mu8    20.77
mu7    21.365
mu6    22.49
mu5    23.59
mu4    24.95
mu3    25.92
mu2    27.19
mu1    28.17
mu0    29.49
p_loaded_0    117.18
p_loaded_1    117.07
p_loaded_2    116.39
p_loaded_3    115.87
p_loaded_4    115.26
p_loaded_5    114.52
p_loaded_6    114
p_loaded_7    113.31
p_loaded_8    112.65
p_loaded_9    112.16
```

```
p_loaded_10   111.34
p_loaded_11   110.87
p_loaded_12   110.27
p_loaded_13   109.48
p_loaded_14   108.87
p_idle_0    107.60
p_idle_1    107.64
p_idle_2    107.72
p_idle_3    107.67
p_idle_4    107.68
p_idle_5    107.54
p_idle_6    107.52
p_idle_7    107.49
p_idle_8    107.23
p_idle_9    107.16
p_idle_10   106.95
p_idle_11   107.01
p_idle_12   106.84
p_idle_13   106.7
p_idle_14   106.68
end
```

```
markov state_6
0_14 1_0    (e^(-1*lambda*u*delta))*
            (1-e^(-1*lambda*(1-u)*delta))
0_14 1_14   (e^(-1*lambda*(1-u)*delta))*
            (1-e^(-1*lambda*u*delta))
0_13 0_14   1-lambda*delta
0_13 1_0    (e^(-1*lambda*u*delta))*
            (1-e^(-1*lambda*(1-u)*delta))
0_13 1_14   (e^(-1*lambda*(1-u)*delta))*
            (1-e^(-1*lambda*u*delta))
0_12 0_13   1-lambda*delta
0_12 1_0    (e^(-1*lambda*u*delta))*
            (1-e^(-1*lambda*(1-u)*delta))
0_12 1_13   (e^(-1*lambda*(1-u)*delta))*
            (1-e^(-1*lambda*u*delta))
0_11 0_12   1-lambda*delta
0_11 1_0    (e^(-1*lambda*u*delta))*
            (1-e^(-1*lambda*(1-u)*delta))
```

```
0_11  1_12   (e^(-1*lambda*(1-u)*delta))*
             (1-e^(-1*lambda*u*delta))
0_10  0_11   1-lambda*delta
0_10  1_0    (e^(-1*lambda*u*delta))*
             (1-e^(-1*lambda*(1-u)*delta))
0_10  1_11   (e^(-1*lambda*(1-u)*delta))*
             (1-e^(-1*lambda*u*delta))
0_9   0_10   1-lambda*delta
0_9   1_0    (e^(-1*lambda*u*delta))*
             (1-e^(-1*lambda*(1-u)*delta))
0_9   1_10   (e^(-1*lambda*(1-u)*delta))*
             (1-e^(-1*lambda*u*delta))
0_8   0_9    1-lambda*delta
0_8   1_0    (e^(-1*lambda*u*delta))*
             (1-e^(-1*lambda*(1-u)*delta))
0_8   1_9    (e^(-1*lambda*(1-u)*delta))*
             (1-e^(-1*lambda*u*delta))
0_7   0_8    1-lambda*delta
0_7   1_0    (e^(-1*lambda*u*delta))*
             (1-e^(-1*lambda*(1-u)*delta))
0_7   1_8    (e^(-1*lambda*(1-u)*delta))*
             (1-e^(-1*lambda*u*delta))
0_6   0_7    1-lambda*delta
0_6   1_0    (e^(-1*lambda*u*delta))*
             (1-e^(-1*lambda*(1-u)*delta))
0_6   1_7    (e^(-1*lambda*(1-u)*delta))*
             (1-e^(-1*lambda*u*delta))
0_5   0_6    1-lambda*delta
0_5   1_0    (e^(-1*lambda*u*delta))*
             (1-e^(-1*lambda*(1-u)*delta))
0_5   1_6    (e^(-1*lambda*(1-u)*delta))*
             (1-e^(-1*lambda*u*delta))
0_4   0_5    1-lambda*delta
0_4   1_0    (e^(-1*lambda*u*delta))*
             (1-e^(-1*lambda*(1-u)*delta))
0_4   1_5    (e^(-1*lambda*(1-u)*delta))*
             (1-e^(-1*lambda*u*delta))
0_3   0_4    1-lambda*delta
0_3   1_0    (e^(-1*lambda*u*delta))*
             (1-e^(-1*lambda*(1-u)*delta))
0_3   1_4    (e^(-1*lambda*(1-u)*delta))*
             (1-e^(-1*lambda*u*delta))
0_2   0_3    1-lambda*delta
0_2   1_0    (e^(-1*lambda*u*delta))*
```

```
                (1-e^(-1*lambda*(1-u)*delta))
0_2  1_3        (e^(-1*lambda*(1-u)*delta))*
                (1-e^(-1*lambda*u*delta))
0_1  0_2        1-lambda*delta
0_1  1_0        (e^(-1*lambda*u*delta))*
                (1-e^(-1*lambda*(1-u)*delta))
0_1  1_2        (e^(-1*lambda*(1-u)*delta))*
                (1-e^(-1*lambda*u*delta))
0_0  0_1        1-lambda*delta
0_0  1_0        (e^(-1*lambda*u*delta))*
                (1-e^(-1*lambda*(1-u)*delta))
0_0  1_1        (e^(-1*lambda*(1-u)*delta))*
                (1-e^(-1*lambda*u*delta))
1_14 0_14       (e^(-1*lambda*delta))*
                (1-e^(-1*mu14*u*delta))
1_14 1_0        (e^(-1*lambda*delta))*
                (e^(-1*mu14*delta))
1_14 2_0        (1-e^(-1*lambda*delta))*
                (e^(-1*mu14*delta))
1_14 0_0        (e^(-1*lambda*delta))*
                (1-(e^(-1*mu14*(1-u)*delta)))
1_13 1_0        (e^(-1*lambda*delta))*
                (e^(-1*mu13*delta))
1_13 2_0        (1-e^(-1*lambda*delta))*
                (e^(-1*mu13*delta))
1_13 0_14       (e^(-1*lambda*delta))*
                (1-(e^(-1*mu13*u*delta)))
1_13 0_0        (e^(-1*lambda*delta))*
                (1-(e^(-1*mu13*(1-u)*delta)))
1_12 1_0        (e^(-1*lambda*delta))*
                (e^(-1*mu12*delta))
1_12 2_0        (1-e^(-1*lambda*delta))*
                (e^(-1*mu12*delta))
1_12 0_13       (e^(-1*lambda*delta))*
                (1-(e^(-1*mu12*u*delta)))
1_12 0_0        (e^(-1*lambda*delta))*
                (1-(e^(-1*mu12*(1-u)*delta)))
1_11 1_0        (e^(-1*lambda*delta))*
                (e^(-1*mu11*delta))
1_11 2_0        (1-e^(-1*lambda*delta))*
                (e^(-1*mu11*delta))
1_11 0_12       (e^(-1*lambda*delta))*
                (1-(e^(-1*mu11*u*delta)))
1_11 0_0        (e^(-1*lambda*delta))*
```

```
                (1-(e^(-1*mu11*(1-u)*delta)))
1_10  1_0       (e^(-1*lambda*delta))*
                (e^(-1*mu10*delta))
1_10  2_0       (1-e^(-1*lambda*delta))*
                (e^(-1*mu10*delta))
1_10  0_11      (e^(-1*lambda*delta))*
                (1-(e^(-1*mu10*u*delta)))
1_10  0_0       (e^(-1*lambda*delta))*
                (1-(e^(-1*mu10*(1-u)*delta)))
1_9   1_0       (e^(-1*lambda*delta))*
                (e^(-1*mu9*delta))
1_9   2_0       (1-e^(-1*lambda*delta))*
                (e^(-1*mu9*delta))
1_9   0_10      (e^(-1*lambda*delta))*
                (1-(e^(-1*mu9*u*delta)))
1_9   0_0       (e^(-1*lambda*delta))*
                (1-(e^(-1*mu9*(1-u)*delta)))
1_8   1_0       (e^(-1*lambda*delta))*
                (e^(-1*mu8*delta))
1_8   2_0       (1-e^(-1*lambda*delta))*
                (e^(-1*mu8*delta))
1_8   0_9       (e^(-1*lambda*delta))*
                (1-(e^(-1*mu8*u*delta)))
1_8   0_0       (e^(-1*lambda*delta))*
                (1-(e^(-1*mu8*(1-u)*delta)))
1_7   1_0       (e^(-1*lambda*delta))*
                (e^(-1*mu7*delta))
1_7   2_0       (1-e^(-1*lambda*delta))*
                (e^(-1*mu7*delta))
1_7   0_8       (e^(-1*lambda*delta))*
                (1-(e^(-1*mu7*u*delta)))
1_7   0_0       (e^(-1*lambda*delta))*
                (1-(e^(-1*mu7*(1-u)*delta)))
1_6   1_0       (e^(-1*lambda*delta))*
                (e^(-1*mu6*delta))
1_6   2_0       (1-e^(-1*lambda*delta))*
                (e^(-1*mu6*delta))
1_6   0_7       (e^(-1*lambda*delta))*
                (1-(e^(-1*mu6*u*delta)))
1_6   0_0       (e^(-1*lambda*delta))*
                (1-(e^(-1*mu6*(1-u)*delta)))
1_5   1_0       (e^(-1*lambda*delta))*
                (e^(-1*mu5*delta))
1_5   2_0       (1-e^(-1*lambda*delta))*
```

```
                (e^(-1*mu5*delta))
1_5  0_6        (e^(-1*lambda*delta))*
                (1-(e^(-1*mu5*u*delta)))
1_5  0_0        (e^(-1*lambda*delta))*
                (1-(e^(-1*mu5*(1-u)*delta)))
1_4  1_0        (e^(-1*lambda*delta))*
                (e^(-1*mu4*delta))
1_4  2_0        (1-e^(-1*lambda*delta))*
                (e^(-1*mu4*delta))
1_4  0_5        (e^(-1*lambda*delta))*
                (1-(e^(-1*mu4*u*delta)))
1_4  0_0        (e^(-1*lambda*delta))*
                (1-(e^(-1*mu4*(1-u)*delta)))
1_3  1_0        (e^(-1*lambda*delta))*
                (e^(-1*mu3*delta))
1_3  2_0        (1-e^(-1*lambda*delta))*
                (e^(-1*mu3*delta))
1_3  0_4        (e^(-1*lambda*delta))*
                (1-(e^(-1*mu3*u*delta)))
1_3  0_0        (e^(-1*lambda*delta))*
                (1-(e^(-1*mu3*(1-u)*delta)))
1_2  1_0        (e^(-1*lambda*delta))*
                (e^(-1*mu2*delta))
1_2  2_0        (1-e^(-1*lambda*delta))*
                (e^(-1*mu2*delta))
1_2  0_3        (e^(-1*lambda*delta))*
                (1-(e^(-1*mu2*u*delta)))
1_2  0_0        (e^(-1*lambda*delta))*
                (1-(e^(-1*mu2*(1-u)*delta)))
1_1  1_0        (e^(-1*lambda*delta))*
                (e^(-1*mu1*delta))
1_1  2_0        (1-e^(-1*lambda*delta))*
                (e^(-1*mu1*delta))
1_1  0_2        (e^(-1*lambda*delta))*
                (1-(e^(-1*mu1*u*delta)))
1_1  0_0        (e^(-1*lambda*delta))*
                (1-(e^(-1*mu1*(1-u)*delta)))
1_0  0_1        (e^(-1*lambda*delta))*
                (1-(e^(-1*mu0*u*delta)))
1_0  0_0        (e^(-1*lambda*delta))*
                (1-(e^(-1*mu0*(1-u)*delta)))
1_0  2_0        (1-e^(-1*lambda*delta))*
                (e^(-1*mu0*delta))
2_0  3_0        1-(e^(-1*lambda*delta))
```

```
2_0 1_0    1-(e^(-1*mu0*delta))
3_0 4_0    1-(e^(-1*lambda*delta))
3_0 2_0    1-(e^(-1*mu0*delta))
4_0 5_0    1-(e^(-1*lambda*delta))
4_0 3_0    1-(e^(-1*mu0*delta))
5_0 6_0    1-(e^(-1*lambda*delta))
5_0 4_0    1-(e^(-1*mu0*delta))
6_0 5_0    1-(e^(-1*mu0*delta))
end
end
var pi_0_sum \
  prob(state_6,0_14)+prob(state_6,0_13)+
  prob(state_6,0_12)+prob(state_6,0_11)+
  prob(state_6,0_10)+prob(state_6,0_9)+
  prob(state_6,0_8)+prob(state_6,0_7)+
  prob(state_6,0_6)+prob(state_6,0_5)+
  prob(state_6,0_4)+prob(state_6,0_3)+
  prob(state_6,0_2)+prob(state_6,0_1)+
  prob(state_6,0_0)
var pi_1_sum \
  prob(state_6,1_14)+prob(state_6,1_13)+
  prob(state_6,1_12)+prob(state_6,1_11)+
  prob(state_6,1_10)+prob(state_6,1_9)+
  prob(state_6,1_8)+prob(state_6,1_7)+
  prob(state_6,1_6)+prob(state_6,1_5)+
  prob(state_6,1_4)+prob(state_6,1_3)+
  prob(state_6,1_2)+prob(state_6,1_1)+
  prob(state_6,1_0)
var utilization \
  (1-pi_0_sum)*100


var k \
  pi_1_sum+(2*prob(state_6,2_0))+
  (3*prob(state_6,3_0))+(4*prob(state_6,4_0))+
  (5*prob(state_6,5_0))+(6*prob(state_6,6_0))


var MRT \
  (k/lambda)*1000


var P1 \
  (p_idle_0*prob(state_6,0_0))+
```

```
  (p_idle_1*prob(state_6,0_1))+
  (p_idle_2*prob(state_6,0_2))+
  (p_idle_3*prob(state_6,0_3))+
  (p_idle_4*prob(state_6,0_4))+
  (p_idle_5*prob(state_6,0_5))+
  (p_idle_6*prob(state_6,0_6))+
  (p_idle_7*prob(state_6,0_7))+
  (p_idle_8*prob(state_6,0_8))+
  (p_idle_9*prob(state_6,0_9))+
  (p_idle_10*prob(state_6,0_10))+
  (p_idle_11*prob(state_6,0_11))+
  (p_idle_12*prob(state_6,0_12))+
  (p_idle_13*prob(state_6,0_13))+
  (p_idle_14*prob(state_6,0_14))
var P2 \
  (p_loaded_0*prob(state_6,1_0))+
  (p_loaded_1*prob(state_6,1_1))+
  (p_loaded_2*prob(state_6,1_2))+
  (p_loaded_3*prob(state_6,1_3))+
  (p_loaded_4*prob(state_6,1_4))+
  (p_loaded_5*prob(state_6,1_5))+
  (p_loaded_6*prob(state_6,1_6))+
  (p_loaded_7*prob(state_6,1_7))+
  (p_loaded_8*prob(state_6,1_8))+
  (p_loaded_9*prob(state_6,1_9))+
  (p_loaded_10*prob(state_6,1_10))+
  (p_loaded_11*prob(state_6,1_11))+
  (p_loaded_12*prob(state_6,1_12))+
  (p_loaded_13*prob(state_6,1_13))+
  (p_loaded_14*prob(state_6,1_14))
var P3 \
  (1-pi_0_sum-pi_1_sum)*p_loaded_0

var P \
  P1+P2+P3
var Pi_2 \
  prob(state_6,2_0)
var Pi_3 \
  prob(state_6,3_0)
var Pi_4 \
  prob(state_6,4_0)
var Pi_5 \
  prob(state_6,5_0)
var Pi_6 \
```

```
  prob(state_6,6_0)
var Pi_sum \
  pi_0_sum+pi_1_sum+prob(state_6,2_0)+
  prob(state_6,3_0)+prob(state_6,4_0)+
  prob(state_6,5_0)+prob(state_6,6_0)
expr pi_0_sum
expr pi_1_sum
expr Pi_2
expr Pi_3
expr Pi_4
expr Pi_5
expr Pi_6
expr Pi_sum
expr k
expr MRT
expr P
expr utilization
end
```

List of Figures

List of Tables

Bibliography

[ADD12] V. Anagnostopoulou, M. Dimitrov, and K. A. Doshi. Sla-guided energy savings for enterprise servers. In *2012 IEEE International Symposium on Performance Analysis of Systems Software*, pages 120–121, April 2012.

[BKZ+15] J. D. Booth, J. Kotra, H. Zhao, M. Kandemir, and P. Raghavan. Phase detection with hidden markov models for dvfs on many-core processors. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 185–195, June 2015.

[BNdM16] Robert Basmadjian, Florian Niedermeier, and Hermann de Meer. Modelling performance and power consumption of utilisation-based dvfs using m/m/1 queues. In *Proceedings of the Seventh International Conference on Future Energy Systems*, e-Energy '16, pages 14:1–14:11, New York, NY, USA, 2016. ACM.

[Bro] Dominik Brodowski. Linux cpufreq-cpufreq governors. `https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt`.

[DB92] Robert Gallager Dimitri Bertsekas. *Data Networks*. Prentice-Hall, 2nd edition, January 1992.

[DM02] D. Hill G. Hinton D. Koufaty J. Miller M. Upton D. Marr, F. Binns. Hyper-threading technology architecture and microarchitecture. Intel Technology Journal, February 14 2002.

[GBT06] H. de Meer G. Bolch, S. Greiner and K. S. Trivedi. *Queuing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications.* WileyBlackwell, 2nd edition, May 2006.

[gKCE+12] S. g. Kim, C. Choi, H. Eom, H. Y. Yeom, and H. Byun. Energy-centric dvfs controling method for multi-core platforms. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 685–690, Nov 2012.

[Hin] Dr. Ralph Hintemann. Energy consumption of data centers continues to increase – 2015 update. `https://www.borderstep.de/wp-content/uploads/2015/01/Borderstep_Energy_Consumption_2015_Data_Centers_16_12_2015.pdf`.

[HPDL15] C. Y. Hsieh, J. G. Park, N. Dutt, and S. S. Lim. Memory-aware co-operative cpu-gpu dvfs governor for mobile games. In *2015 13th IEEE Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia)*, pages 1–8, Oct 2015.

[JKL+13] H. Beom Jang, J. M. Kim, H. J. Lee, S. W. Chung, Y. Shin, and J. C. Son. Intelligent governor for low-power mobile application processors. In *2013 International SoC Design Conference (ISOCC)*, pages 206–207, Nov 2013.

[Kar13] M. Karpowicz. On the design of energy-efficient service rate control mechanisms: Cpu frequency control for linux. In *2013 24th Tyrrhenian International Workshop on Digital Communications - Green ICT (TIWDC)*, pages 1–6, Sept 2013.

[Kid14] Taylor IoT Kidd. Power management states: P-states, c-states, and package c-states. `https://software.intel.com/en-us/articles/power-management-states-p-states-c-states-and-package-c-states#_Toc383778910`, 17.4. 2014.

[Lit] John D.C. Little. A proof for the queuing formula: L= λ w. Operations Research, Volume 9, Issue 3 (May - Jun., 1961), 383-387.

[Pal] Venkatesh Pallipadi. Linux cpufreq-stats driver. `https://www.kernel.org/doc/Documentation/cpu-freq/cpufreq-stats.txt`.

[SSS+] Arman Shehabi, Sarah Josephine Smith, Dale A. Sartor, Richard E. Brown, Magnus Herrlin, Jonathan G. Koomey, Eric R. Masanet, Nathaniel Horner, Inês Lima Azevedo, and William Lintner. United states data center energy usage report. `https://eta.lbl.gov/publications/united-states-data-center-energy`, 06/2016.

[Tor08] Gabriel Torres. Everything you need to know about the cpu c-states power saving modes. `http://www.hardwaresecrets.com/everything-you-need-to-know-about-the-cpu-c-states-power-saving-modes/`, 05. 9. 2008.

[VP] Alexey Starikovskiy Venkatesh Pallipadi. The ondemand governor-past, present, and future. `https://www.kernel.org/doc/ols/2006/ols2006v2-pages-223-238.pdf`.

[web] Intel p-state driver. `https://www.kernel.org/doc/Documentation/cpu-freq/intel-pstate.txt`.