Task Assignment in Server Farms under Realistic Workload Conditions

A thesis submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

Malith Jayasinghe B.E. (Hons.),
School of Computer Science and Information Technology,
College of Science, Engineering and Health,
RMIT University,
Melbourne, Victoria, Australia.

December 2011

Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; any editorial work, paid or unpaid, carried out by a third party is acknowledged; and, ethics procedures and guidelines have been followed.

Malith Jayasinghe
School of Computer Science and Information Technology
RMIT University

Acknowledgments

First and foremost I would like to thank my primary supervisor, Professor Zahir Tari. I appreciate all his support, guidance, ideas, and funding to make my PhD experience productive and stimulating.

I would like to thank my second supervisor, Professor Panlop Zeephongsekul. The joy and enthusiasm he has for research was contagious and encouraged me immensely. I am extremely grateful for the time he spent working with me on complex queueing problems.

I sincerely thank, Dr. James Broberg, my consultant during my first year, for his support and advice.

I gratefully acknowledge the funding sources that made my PhD work possible. I was supported by a School of Computer Science and IT Research Scholarship during my first year and the remaining years by the RMIT University Postgraduate Research Scholarship.

I wish to thank my fellow PhD students and the staff members of School of Computer Science and IT for their companionship and collegiality.

I would like to thank my family for all their love and encouragement. I am grateful to my parents for raising me with a love of science and supporting me in all my pursuits, and my sister for her love and her good humour. I am grateful to my grandfathers who cared and supported me immensely during my childhood.

I would like to extend my sincere thanks and gratitude to my wife's family, my sister-in-law and my brother-in-law for their love and support.

Most of all I wish to thank my loving, supportive, encouraging, and patient wife Chathuri for her unfailing support particularly during the final stages of my PhD.

Malith Jayasinghe
RMIT University
December 2011.

This thesis is dedicated to

my loving grandmother

who passed away on December 29, 2007. She loved me endlessly and devoted her life to make sure that I had a good life.

Credits

Portions of the material in this thesis have previously appeared in the following publications:

1. Malith Jayasinghe, Zahir Tari, Panlop Zeephongsekul, Albert Y. Zomaya, Task Assignment in Server Farms when the Service Time Distribution of Tasks is not known *a Priori*, IEEE Transactions on Parallel and Distribute Systems (Under submission).

2. Malith Jayasinghe, Zahir Tari, Panlop Zeephongsekul, Albert Y. Zomaya, Task Assignment in Multiple Server Farms using Preemptive Migration and Flow Control, Journal of Parallel and Distributed Computing (JPDC), 71(12), pages 1608 – 1621, 2011.

3. Zahir Tari, Ann Khoi Anh Phan, Malith Jayasinghe and Vidura Gamini Abhaya, On the Performance of Web Services, Springer, USA, 2011

4. Malith Jayasinghe, Zahir Tari, and Panlop Zeephongsekul. A Scalable Multi-tier Task Assignment Policy with Minimum Excess Load, International Symposium on Computers and Communication (ISCC), pages 913 – 918, 2010.

5. Malith Jayasinghe, Zahir Tari, and Panlop Zeephongsekul. Performance Analysis of Multi-level Time Sharing Task Assignment Policies on Cluster-based Systems, International Conference on Cluster Computing (IEEE CLUSTER), pages 265 – 274, 2010.

6. Malith Jayasinghe, Zahir Tari, and Panlop Zeephongsekul. Multi-level Multi-server Task Assignment with Work-conserving Migration, 9th IEEE International Symposium on Network Computing and Applications (IEEE-NCA), pages 178 – 181, 2010

7. Malith Jayasinghe, Zahir Tari, Panlop Zeephongsekul, and James Broberg. On the Performance of Multi-level Time Sharing Policy under Heavy-tailed Workloads, International Symposium on Computers and Communication (ISCC), pages 956 – 962, 2009.

8. Malith Jayasinghe, Zahir Tari, and Panlop Zeephongsekul. The Impact of Quanta on the Performance of Multi-level Time Sharing Policy under Heavy-tailed Workloads. Proceedings of the Thirty-Second Australasian Conference on Computer Science (ACSC), 91, pages 83 – 91, 2009.

Contents

List of Figures

List of Tables

Abstract

Server farms have become very popular in recent years since they effectively address the problem of large delays, a common problem faced by many organisations whose systems receive high volumes of traffic. Recently, there has been a wide use of these server farms in two main areas, namely, Web hosting and scientific computing. The performance of such server farms is highly reliant on the underlying task assignment policy, a specific set of rules that defines how the incoming tasks are assigned to and processed at hosts. The aim of a task assignment policy is to optimise certain performance criteria such as the expected waiting time, slowdown or flow-time. One of the key factors that affect the performance of these policies is the service time distribution of tasks. There is extensive evidence indicating that the service times (processing times) of modern computer workloads closely follow heavy-tailed distributions that possess high variance. However, in certain environments, the service time distributions of tasks are unknown. Imposing parametric assumptions (e.g. heavy-tailed, exponential, etc.) in such cases can lead to inaccurate and unreliable inferences.

Traditional task assignment policies such as Random and Round-Robin do not perform well under realistic workload conditions (e.g. heavy-tailed), because they have not been designed to optimise the performance under such workload conditions. Considerable efforts have been made in recent years to devise more efficient policies. These policies use special techniques (such as unbalancing the load among hosts and reducing the task size variability in hosts queues, etc.) to improve the performance. Although these policies perform well in certain environments under specific workload conditions, they have several major limitations. These include the assumption of known service times, inability to efficiently assign tasks in time sharing server farms, poor performance under changing workload conditions and poor performance under multiple server farms.

This thesis aims at proposing novel task assignment policies for assigning tasks in server farms under two main classes of realistic workload conditions, namely, the heavy-tailed and arbitrary service time distributions. Arbitrary service time distributions are assumed, for cases where the underlying service time distribution of tasks is unknown. Under such conditions, the design of policies is

an extremely challenging problem due to the complexity involved in the analytical modelling. In this thesis we employ stochastic modelling techniques to model the performance of systems and to guide the design of new policies to optimise the performance.

We focus on devising task assignment policies that can assign (or schedule) dynamic web content (e.g. databases requests, various scripts, etc.) and scientific workloads (e.g. complex algorithms, mathematical models, etc.), because there has been a rapid increase in these workloads in recent years. For such workloads, the server is typically the bottleneck resource (as opposed to the bandwidth), making server side scheduling more important. Service times of much dynamic web content and scientific workloads are not known *a priori* and very difficult to estimate.

The first problem addressed in this thesis is how to efficiently schedule tasks in a time sharing server under heavy-tailed service time distributions. Scheduling is the fundamental way to minimise response times of tasks and even a small change to the scheduling policy can result in massive improvements in the performance. We concentrate on a particular scheduling policy called, multi-level time sharing policy (MLTP), which performs well under distributions with the property of decreasing failure rate, a key property of modern heavy-tailed traffic. Existing performance models related to MLTP are based on very unrealistic conditions such as exponential service time distributions, infinite number of levels and infinitely small quanta. We derive a performance model for MLTP under finite levels when the quanta are not infinitely small and investigate the behaviour of MLTP under various workload conditions. We show that MLTP with optimal quanta can result in significant performance improvements over other policies under a range of workload conditions.

The second problem addressed in the thesis is how to design efficient task assignment policies for time sharing server farms. Existing policies have been mainly designed for assigning tasks in batch computing server farms and therefore, they perform poorly under time sharing server farms. We propose three task assignment policies called, MLMS, MLMS-M and MLMS-PM dedicated to time sharing server farms. The core features of these policies include the global and local reduction of task size variance, provision of preferential treatment to small tasks and task migration between hosts. These features enable these policies to perform well under a range of workload conditions. For example, MLMS-M with five levels outperforms TAGS by a factor of 6.75 under high system loads and high task size variabilities.

The third problem addressed in this thesis is the way to efficiently assign tasks in multiple server farms under heavy-tailed service time distributions. Existing task assignment policies are not very efficient in assigning tasks in multiple server farms, because they have not been designed to exploit the properties of such environments. We propose MCTPM for a multiple server farm, which is based on a flexible multi-tier host architecture. MCTPM controls the traffic flow into server farms via a

global dispatching device so as to optimise the performance. In addition, it supports preemptive task migration between servers in the same farm as well as between servers in different farms. The performance analysis of MCTPM shows that it outperforms other policies under a range of scenarios. For example, MCTPM outperforms MC-TAGSPM by a factor of 5 under moderate system loads and low task size variabilities.

The last problem investigated in thesis is the way to design adaptive task assignment policies that make no assumptions about the underlying service time distribution of tasks. We propose a novel task assignment policy, called ADAPT-POLICY, which is based on multiple static-based task assignment policies. ADAPT-POLICY defines a set of policies for the server farm based on the specific properties of the farm. These policies are selected in such a way that they have different performance characteristics under different workload conditions (i.e. service time distributions). ADAPT-POLICY adaptively changes the task assignment policy to suit the most recent traffic conditions. The experimental performance analysis of ADAPT-POLICY shows that it outperforms other policies under a range of evolving traffic conditions.

Chapter 1

Introduction

The rapid growth of the Internet over the last ten years has resulted in a rapid increase in the amount of traffic reaching web servers. According to the traffic-analysis surveys, Internet traffic has doubled every year since 1997 until 2003 [Odlyzko, 2003]. The Internet continues to grow at phenomenal rates due to emergence of new technologies such as Web 2.0 [Krogfoss et al., 2011]. This has resulted in a significant increase in the demand for computational resources.

Distributed systems offer a cost-effective and scalable solution to the increasing service demands placed on servers. One such popular distributed system model is called the server farm model (cluster-based distributed system), which consists of a set of loosely coupled hosts offering mirrored services and hosting replicated content. Figure 1.1 illustrates the host architecture of a typical web server farm. Server farms have become very popular in recent years since they effectively address the problem of large delays, a common problem faced by many organisations whose systems receive high volumes of traffic. Recently, there has been a wide use of these server farms in two main areas, namely, Web hosting and scientific computing. The performance of such server farms is highly reliant on the underlying task assignment policy, a specific set of rules that defines how the incoming tasks are assigned to and processed at back-end hosts. The aim of a task assignment policy is to optimise certain performance criteria such as the expected waiting time, slowdown or flow-time.

Considerable efforts have been made in recent years to devise task assignment policies that can efficiently assign tasks in server farms. Unfortunately, these policies possess major limitations such as the assumption of known service times [Zhang and Sun, 2005; Ciardo et al., 2001; Harchol-Balter et al., 1999; Tari et al., 2005; Fu and Tari, 2003], inability to efficiently assign tasks in time sharing systems [Harchol-Balter, 2002; Broberg et al., 2004; 2006; Ciardo et al., 2001; Harchol-Balter et al., 1999; Tari et al., 2005; Fu and Tari, 2003], poor performance under changing workload con-

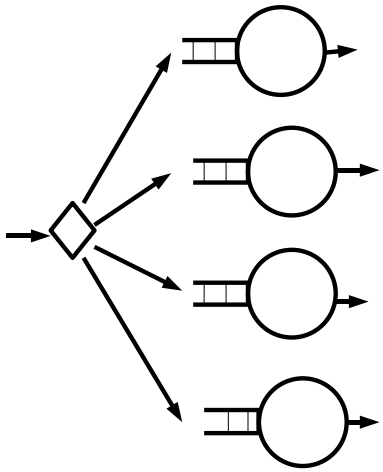


Figure 1.1: A cluster-based distributed computing system

ditions [Harchol-Balter, 2002; Broberg et al., 2004; 2006; Ciardo et al., 2001; Harchol-Balter et al., 1999; Tari et al., 2005; Fu and Tari, 2003], poor performance under multi-cluster distributed systems [Harchol-Balter, 2002; Broberg et al., 2004; 2006; Zhang and Sun, 2005; Ciardo et al., 2001; Harchol-Balter et al., 1999; Tari et al., 2005; Fu and Tari, 2003] and the assumption of exponential service time distributions [Zhang and Fan, 2008]. These issues are discussed in detail in Section 1.1.

Devising efficient task assignment policies is an extremely challenging problem, particularly under realistic workload conditions because under such conditions, it is very difficult to analytically model the performance of task assignment policies. Under many contexts (i.e. workload conditions), there exists no 'optimal' task assignment policy for assigning tasks. This has led to the development of many task assignment policies and the following associated **open** problem:

OPEN PROBLEM: Is there an optimal task assignment policy for assigning tasks in server farms under a specific set of conditions?

This thesis aims at proposing novel task assignment policies to assign tasks in server farms under realistic computer workload conditions. Although computing workloads have numerous characteristics, the most important characteristic, which has a direct impact on the performance is the service time distribution of tasks (jobs). In this thesis we focus on the following two realistic workload scenarios.

1. **Heavy-tailed service time distributions**: There is extensive evidence indicating that the service times (processing times) of modern computer workloads follow heavy-tailed distribu-

tions that possess high variance [Arlitt and Williamson, 1996; Barford et al., 1999; Arlitt and Williamson, 1997; Barford and Crovella, 1998; Crovella et al., 1998b; Crovella and Bestavros, 1997; Harchol-Balter and Downey, 1997; Willinger et al., 1997; Karagiannis et al., 2004; Arlitt and Jin, 2000; Cheng et al., 2010; Markovich, 2011; Loiseau et al., 2011]. The traditional assumption was that the processing requirements are exponentially distributed. Heavy-tailed distributions differ from exponential distributions in numerous ways. These distributions have thicker tails compared to those of exponential distributions and unlike exponential distributions, these do not possess traits such as constant failure rate and memoryless property of the exponential distribution.

2. **Arbitrary service time distributions**: An arbitrary service time distribution simply means that no assumptions are made regarding the underlying service time distribution of tasks. Although the heavy-tailed service time distributions have been justified for a majority of cases, for certain other cases, there is no evidence to show that the service times of tasks can be represented using a particular type of distribution [Zhang and Sun, 2005] due to the following two main reasons.

   (a) Service times of certain tasks are not always recorded. As such, it is difficult to find a sufficient number of data sets, which can be used to fit a probability distribution to the entire population of task sizes.

   (b) Even if such data sets are available, they may come from heterogeneous family of distributions and any attempt to fit a particular distribution to it would be impossible. Moreover, there is also a possibility for the service time distribution of tasks to vary over time due to the non-stationary nature of traffic [j. Lin et al., 2006; Bertsimas and Mourtzinou, 1997; Zhang et al., 2003; Zhang and Sun, 2005].

It is important to point out that the task assignment policies that can handle arbitrary service time distributions may be used for assigning tasks under heavy-tailed service time distributions as well. However, if there is clear evidence [Arlitt and Williamson, 1996; Barford et al., 1999; Arlitt and Williamson, 1997; Willinger et al., 1997; Markovich, 2011] to show that the service times of tasks follow heavy-tailed distributions, this is not recommended due the following two reasons.

1. The generic task assignment policies (i.e. policies that can handle arbitrary service time distributions) may be computationally intensive.

2. Specific techniques may be utilised to further improve the performance if there is prior knowledge about the presence of heavy-tailed service time distributions.

Throughout this thesis we assume that the service times of tasks are not known *a priori*. The particular focus on such tasks is because the recent advances in the Web has resulted in a rapid growth in the dynamic content [Krogfoss et al., 2011; Group, 2009]. Moreover, a significant growth is observed in the amount of scientific workloads being generated. This is justified by the increasing number of distributed computing facilities being built around the world specifically to serve scientific workloads. Service times of much scientific workloads and dynamic web content are rarely known in advance and are difficult to estimate. In addition, for such workloads, the server is typically the bottleneck resource as opposed to the bandwidth and therefore, scheduling of tasks at the server side plays an important role when it comes to the performance.

There are two main metrics used to evaluate the performance of task assignment policies, namely, the expected waiting time and expected slowdown, the expected waiting time being the most commonly used performance metric. Slowdown is defined as the ratio between a task's waiting time and its service time and it measures the fairness of a scheduling policy under a given task assignment policy. Indeed, the aim is to minimise the value of these performance metrics. In this thesis we use the expected waiting time as the main performance metric. In Chapter 3, however, we use both the expected waiting time and expected slowdown.

The rest of this chapter is organised as follows. Section 1.1 summarises the main limitation of existing task assignment policies. Section 1.2 presents the research questions addressed in this thesis followed by the key contributions of this thesis are presented in Section 1.3. The structure of the rest of this thesis is summarised in Section 1.4.

1.1 Limitations of existing work

We have identified the following five main limitations in existing work.

1. Several existing solutions assume that the sizes of tasks (i.e. processing requirement) are known *a priori* or can be estimated *a priori* [Harchol-Balter et al., 1999; Ciardo et al., 2001; Tari et al., 2005; Zhang and Sun, 2005]. Although this assumption reduces the complexity of the task assignment problem significantly, such solutions can only be used for assigning particular types of tasks, in particular static web content. For many other types of computer workloads (such as dynamic content and scientific workloads), it is not possible to estimate the service times prior to execution. As discussed, devising efficient task assignment policies for assigning such tasks (i.e. dynamic web content and scientific workloads) is important because there has been a rapid increase in these type of workloads in recent years [Krogfoss et al., 2011; Group, 2009].

2. Existing task assignment policies do not scale well [Harchol-Balter, 2002; Broberg et al., 2004], particularly if the number of servers in the farm is relatively high. This means that the performance of these policies degrades as the number of servers increases. The main reason for this is that these existing task assignment policies restart certain tasks from scratch. This results in a significant excess load on the system, which in turn degrades the performance of the system.

3. Existing task assignment policies that can efficiently assign tasks under heavy-tailed service time distributions have been designed for batch computing systems. These policies serve tasks either in a first-come-first-served (FCFS) manner until completion or up to a predefined time limit [Harchol-Balter, 2002; Broberg et al., 2004; Psounis et al., 2005; Broberg et al., 2006]. Very little work has been done to improve the performance of time sharing server farms under heavy-tailed service time distributions. Moreover, much existing work on time sharing policies investigates only the performance in a single server system. These works are based on very unrealistic assumptions, namely, the exponential service time distributions, infinitely small quantum and infinite number of priority classes [Aalto, 2006; Aalto et al., 2007; 2004]. Neither infinitely small quantum nor infinite number of priority levels can be implemented on real computer systems.

4. Existing task assignment policies [Harchol-Balter, 2002; Broberg et al., 2004; 2006; Zhang and Sun, 2005; Ciardo et al., 2001; Harchol-Balter et al., 1999; Tari et al., 2005; Fu and Tari, 2003] proposed for stand-alone server farms are not efficient in multiple server farm environments because these have not been designed to exploit the properties of such environments. As such, existing task assignment policies do not perform well under multiple server farm environments.

5. Existing task assignment policies cannot be used when the service time distribution of tasks is not known *a priori*. As was pointed out, although the heavy-tailed service distributions have been observed extensively, for certain environments, heavy-tailed or another parametric type of distribution cannot be assumed due the lack of evidence for the presence of such distributions [Zhang and Sun, 2005]. Under such conditions, existing task assignment policies [Harchol-Balter, 2002; Broberg et al., 2004; 2006; Zhang and Sun, 2005; Ciardo et al., 2001; Harchol-Balter et al., 1999; Tari et al., 2005; Fu and Tari, 2003] cannot efficiently assign tasks, as they are based on particular types of service time distributions such as Pareto, exponential and log-normal.

1.2 Research questions

The problem of task assignment in server farms has been extensively studied over the last several years. Yet existing work possesses major limitations as described in Section 1.1. With ample evidence showing that service time distributions of most tasks no longer follow exponential service time distributions, there is an urgent need to devise more efficient task policies that can handle non-traditional (i.e. non-exponential) workloads. Taking into account the limitations of existing work, we have addressed and solved the following four important research questions.

1. How can we efficiently schedule tasks in a time sharing server under heavy-tailed service time distributions?

   In this research question we investigate the way to improve the performance in a time sharing server under heavy-tailed workloads. We concentrate on a particular scheduling policy called, multi-level time sharing policy (MLTP), as there is evidence indicating that MLTP can result in significant performance improvements over other policies, if the service time distribution of tasks possesses the property of decreasing failure rate [Aalto et al., 2004; 2007], a key property of modern heavy-tailed distributions. MLTP gives preferential treatment to small tasks (i.e. tasks with small service times) as well as reduces the variability of tasks sizes in queues. In doing so, MLTP improves the performance (i.e. expected waiting time, slowdown, etc.) of small tasks, which in turn improves the overall performance of the system under heavy-tailed service time distributions. Note that under heavy-tailed service time distributions, probability of small tasks appearing is very high, while the probability of large tasks appearing is very low.

   Much existing work on MLTP has been carried out under very unrealistic conditions such as infinitely small quanta, infinite number of levels and exponential service time distributions. We investigate the performance of MLTP, under heavy-tailed workloads (service time distributions) under finite number of levels, when the quanta are not infinitely small quanta. Such a policy is more consistent with those implemented on real computer systems and the findings will enable system designers to better understand how the factors such as the quanta, system load, number of levels and task size variability will affect the performance of MLTP.

2. How can we efficiently assign tasks in time sharing server farms under heavy-tailed workload conditions?

   Here we investigate the way to improve the performance in time sharing server farms under heavy-tailed workloads. Many existing task assignment policies [Tari et al., 2005; Harchol-Balter et al., 1999; Harchol-Balter, 2002; Broberg et al., 2004; 2006; Zhang and Sun, 2005;

Ciardo et al., 2001] have been designed for batch computing server farms that process tasks either in a FCFS manner until completion or up to a predefined time limit. The expected waiting time of a task in a FCFS queue is proportional to the variance of the service time distribution [Kleinrock, 1975]. As the heavy-tailed service time distributions possess very high variance, the expected waiting time of these policies are not satisfactory under heavy-tailed service time distributions. We focus on designing efficient task assignment policies specifically for time sharing systems taking into consideration the specific properties of time sharing systems (e.g. task preemption facilities).

3. How can we efficiently assign tasks in multiple batch server farms under heavy-tailed workload conditions?

   This research question investigates the way to improve the performance in multiple server farms by devising more efficient task assignment policies. Existing task assignment policies [Harchol-Balter, 2002; Broberg et al., 2004; 2006; Ciardo et al., 2001; Harchol-Balter et al., 1999; Tari et al., 2005; Fu and Tari, 2003] are not very efficient in assigning tasks in multiple server farms because they have not been designed to exploit the properties of such environments. With the availability of high speed networks (e.g. a fibre optics network can provide maximum data transfer rates of more than 100 Gbps) and operating systems with specific properties such as preemptive migration, there exist many windows of opportunity to design more efficient task assignment policies for assigning tasks in multiple server farms. Such policies can better utilise the resources in multiple server farm environments and therefore, can perform better compared to those that optimise the performance in stand-alone server farms.

4. How can we efficiently assign tasks in server farms when the service time distribution of tasks is not known *a priori*?

   The previous three research questions pertain to performance optimisation under heavy-tailed workloads. In this research question we investigate the way to design adaptive task assignment policies, which makes no assumptions regarding the underlying service time distribution of tasks. As was pointed out, many existing policies [Harchol-Balter, 2002; Broberg et al., 2004; 2006; Ciardo et al., 2001; Harchol-Balter et al., 1999; Tari et al., 2005; Fu and Tari, 2003] assume particular (parametric) service time distribution (e.g. Pareto, Exponential, etc.) and a set of fixed (static) parameters for that particular parametric distribution. Due to these assumptions, these policies can easily compute their optimal scheduling parameters (server cut-offs, etc.) off-line so as to optimise a given performance criteria such as the expected waiting time

and expected slowdown. The main issue with these approaches is that since their scheduling parameters are computed off-line to optimise the performance under a specific scenario, these approaches [Harchol-Balter, 2002; Broberg et al., 2006; 2004; Harchol-Balter et al., 1999] cannot respond to variations that occur in the incoming service time distribution. As such, performance of these policies degrades under constantly evolving operational conditions.

1.3 Contribution

1. How can we efficiently schedule tasks in a time sharing server under heavy-tailed service time distributions?

   We devote our attention to a particular time sharing policy called, Multi-level time sharing Policy (MLTP). There are two main reasons for this: 1) MLTP requires no prior knowledge about actual task sizes and therefore, it can be used for scheduling a wide range of task types including dynamic web content and scientific workloads and 2) MLTP has shown significant performance improvements under distributions with the property of decreasing failure rate [Aalto et al., 2004; 2007], a key property of modern traffic represented by heavy-tailed distributions.

   MLTP consists of multiple queues (levels). Each new task that arrives at the system is placed at the lowest level (queue), where the task is served in a FCFS manner until it receives maximum of amount of service called, Quantum 1. If the service time of the task is less than or equal to Quantum 1, the task departs the system. Otherwise, the task is placed at the second queue, where the task is serviced in a FCFS manner until it receives at most Quantum 2, where Quantum 2 represents the maximum amount of service a task can receive in the second queue. The task propagates through the system of queues until the total processing time the task has so far received is equal to its service time at which point it leaves the system. Via its multi-level queueing model, MLTP speeds up the flow of small tasks. This results in significant performance improvements under heavy-tailed service time distributions.

   Existing work that investigates the performance of MLTP are based on the assumptions that the number of levels are infinite and the quanta are infinitely small [Aalto et al., 2007; 2005; 2004], because under these conditions it is relatively easy to model the behaviour of tasks under MLTP. However, neither infinite levels nor infinitely small quanta are practical or ever feasible to implement on real computer systems. A handful of studies that investigate the performance MLTP under positive quanta are based on the assumption that both the inter-arrival times and service times follow exponential distributions [Coffman and Kleinrock, 1968].

We devise a performance model for MLTP without assuming infinite number of levels and infinitely small quanta. Using this performance model, we investigate the performance of MLTP under a range of workload scenarios. First we show that MLTP with optimal set of quanta (MLTP-O) significantly outperforms both the MLTP with equal quanta (MLTP-E) and FCFS, especially when the system load and variability of service times are high. Second we investigate the impact of number of levels on the performance and show that as the number of levels increases, the performance (i.e. expected waiting time and expected slowdown) of both MLTP-O and MLTP-E improves. We investigate the behaviour of quanta for the case of two queues. We show that under a given workload condition, the set of quanta that will result in the best performance is unique for almost all the workload conditions. In addition, we investigate the effect of overestimating and underestimating the optimal quanta and discuss the measures that can be taken to minimise the performance degradation due to overestimating or underestimating the optimal set of quanta. Finally, we investigate the performance of MLTP-E under a large number of queues. For both performance metrics, we show that the relationship between the performance and the number of levels has a power law relationship and the coefficients of the power curve are functions of both the variability of tasks and the system load.

2. How do we efficiently assign tasks in time sharing server farms under heavy-tailed workload conditions?

   We propose three novel task assignment policies that are suitable for assigning tasks in three different types of time sharing server farms. These policies are called, Multi-level Multi-server Task Assignment Policy (MLMS), Multi-level Multi-server Task Assignment Policy based on Task Migration (MLMS-M) and Multi-level Multi-server Task Assignment Policy based on Preemptive Task Migration (MLMS-PM). These policies improve the performance first by giving preferential treatment to small tasks and second by reducing the task size variability in queues. MLMS reduces the variability of tasks locally (i.e. within hosts), while MLMS-M and MLMS-PM utilise both local (within hosts) and global (host level) task size variance reduction mechanisms. The local variance reduction is accomplished by using MLTP to schedule tasks, while the host level variance reduction (global) is accomplished by supporting preemptive and non-preemptive task migration between hosts. Task migration allows the tasks with similar sizes to be processed at the same host by migrating tasks with similar sizes into the same host. This improves the performance under heavy-tailed service time distributions, because it reduces the variance of task sizes in host queues. MLMS-PM is based on preemptive migration,

while MLMS-M facilities non-preemptive migration.[1]

The analytical performance analysis of MLMS indicates that MLMS outperforms recent task assignment policies under certain conditions. For example, under high system loads and high task size variabilities, MLMS with twenty levels outperforms TAGS (Task Assignment based on Guessing Size) [Harchol-Balter, 2002] by a factor of 5. The analytical performance analysis of MLMS-M and MLMS-PM indicates that these policies significantly outperform recent task assignment policies. For example, MLMS-M with five levels outperforms TAGS by a factor of 6.75 under highly variable heavy-tailed workloads and high system loads. Under the same conditions, MLMS-PM with 5 levels outperforms TAGS-PM [Broberg et al., 2006; Harchol-Balter, 2002] by a factor of 4. The improvement in the performance depends on the variability of traffic, system load, number of levels and number of hosts. The most significant improvement (in the performance) is noticed under very high task size variabilities and high system loads. The performance of MLMS-M improves with the number hosts for certain task size variabilities, while the performance MLMS-PM improves with the number of hosts for all the cases considered.

3. How can we efficiently assign tasks in multiple batch server farms under heavy-tailed workload conditions?

   First we propose Multi-tier Task Assignment Policy with Minimum Excess Load (MTTMEL) for a stand-alone batch server farm. This policy addresses the core limitations of existing task assignment policies (e.g. poor performance under high task size variabilities, poor performance under low and moderate task size variabilities, poor performance under high system loads, poor performance in large-sized server farms, etc.). MTTMEL is based on a flexible multi-tier host architecture, where the hosts in tiers only process tasks whose sizes are within a certain size range. By grouping and processing tasks in such a manner MTTMEL reduces the variance of tasks in host queues, and this leads to significant performance improvements. This multi-tier host architecture of MTTMEL offers a high degree of flexibility in terms of the number of tiers as well as the hosts to be used in server farms. These parameters (i.e. number of tiers and number of hosts) can be computed to optimise the performance under a given workload scenario (e.g. a system load and task size variability). Furthermore, this multi-tier host architecture speeds up the flow of small tasks by processing the small tasks in a relatively

[1] Under preemptive migration, information about current status of the task such as the process address space and register content are migrated from the source host to the destination host, which resumes the execution of the task. Non-preemptive migration does not require the current status of the running tasks to be migrated from the source host to the destination node.

large number of hosts. This minimises the expected waiting time of small tasks, which leads to an improvement in the overall performance.

Second we extend MTTMEL and propose Multi-Cluster Task Assignment based on Preemptive Migration (MCTPM) for assigning tasks in multiple server farms. MCTPM is based on the same multi-tier host architecture introduced in MTTMEL. MCTPM controls the traffic flow into server farms via a global dispatching device so as to optimise the performance. MCTPM also supports preemptive task migration between servers in the same farm as well as between servers in different farms. Preemptive migration feature of MCTPM ensures that MCTPM can resume the execution of a task that was previously suspended at a different host.

The experimental and analytical performance analysis of MCTPM shows that it significantly outperforms both the traditional and recent policies under a wide range of workload conditions. For example, MCTPM outperforms MC-TAGSPM [Harchol-Balter, 2002] by a factor of 5 under moderate system loads and low task size variabilities.

4. How do we efficiently assign tasks in server farms when the service time distribution of tasks is not known *a priori*?

   We propose an adaptive task assignment policy, called ADAPT-POLICY, which is based on the concept of multiple static-based task assignment policies. ADAPT-POLICY defines a set of policies for a given distributed system taking into account the specific properties of the system. These policies are selected in such a way that they have different performance characteristics under different workload conditions (i.e. service time distributions, etc.). The objective is to use the task assignment policy with the best performance (i.e. the one with the least expected waiting time) to assign tasks. Which task assignment policy performs the best depends on the traffic conditions that vary over time. ADAPT-POLICY determines the best task assignment using the service time distribution of tasks (and various other traffic properties), which is estimated on-line and then it adaptively changes the task assignment policy to suit with the most recent traffic conditions.

   ADAPT-POLICY consists of three main stages, namely, the on-line data collection, on-line density estimation and on-line selection of task assignment policies. The aim of on-line data collection is to collect the service times of tasks that are needed to estimate the service time distribution and its distributional properties. In the density estimation stage probability density function, cumulative density function and moments of the probability density function are estimated. The technique ADAPT-POLICY uses to estimate these distributions, and their

properties are called, the non-parametric kernel-based density function estimation [Wand and Jones, 1995]. These techniques do not impose many restrictions on the underlying probability distributions, they are therefore, considered a much more general approach to estimation with a wider range of validity than the corresponding parametric method of estimation such as method of moments and maximum likelihood estimation [Wand and Jones, 1995; Mood et al., 1974]. In the last stage ADAPT-POLICY determines the best task assignment policy to assign the next batch tasks and then adaptively changes the task assignment accordingly. Simulation results show that ADAPT-POLICY outperforms other task assignment policies (e.g. ADAPT-TAGS [Harchol-Balter, 2002]), under a wide range of scenarios.

1.4 Organisation of thesis

Chapter 2 provides a detail discussion of related work. The first research question is investigated in Chapter 3, while Chapter 4 investigates the second research question. Chapter 5 deals with the third research question and Chapter 6 addresses the last research question. The thesis is concluded in Chapter 7.

Chapter 2

Background and Related Work

The aim of this chapter is twofold. First it provides some background information required to understand the rest of this thesis. Second it provides an overview of some important existing work related to the new contributions made in this thesis. As discussed, this thesis proposes novel task assignment policies that can efficiently assign tasks in server farms under two main classes of workloads, namely, the heavy-tailed and arbitrary service time distributions. The performance models for these policies are developed by applying queuing theoretic modelling techniques. Therefore, to understand these solutions, it is important to have an understanding of queueing theoretic modelling and workload properties. These are discussed in Section 2.1.

The performance of server farms is highly dependent on both the scheduling policy (used to schedule tasks at back-hosts) as well as the task assignment policy (used for assigning tasks into back-end hosts). Section 2.2 provides some details about well known scheduling policies. Sections 2.3 and 2.4 discuss traditional and recent task assignment policies respectively. Chapter is concluded in Section 2.5.

2.1 Background

This section provides some background knowledge required to understand the rest of this thesis. First we will give a brief introduction to the queuing theoretic modelling that is used throughout this thesis to model the performance of task assignment policies. Second we discuss the properties of heavy-tailed workloads and introduce the Bounded Pareto distribution, which is used to represent heavy-tailed workloads throughout this thesis. Third we discuss how to represent task arrivals into systems. Finally, we give a brief overview of distributed systems model that we consider in this thesis.

2.1.1 Introduction to queuing theory

Throughout this thesis we utilise queueing theoretic principles to model the performance of various task assignment policies. Queueing theory provides a stochastic and probabilistic approach to investigate the operation of queues. Figure 2.1 illustrates a basic queueing process.

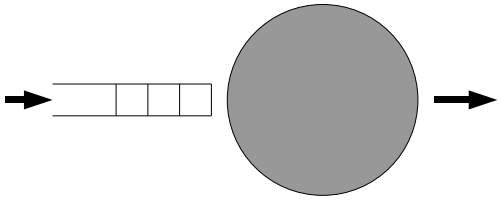


Figure 2.1: A queueing process

Kendall's notation [Kendall, 1953] is used to describe a queueing system. Kendall's notation represents a queueing system in a form of $A/B/C/D/E$, where A, B, C, D and E represent the arrival pattern of tasks, service pattern of tasks, number of servers, system capacity and service discipline respectively.

The arrival pattern describes the distribution of inter-arrival times of tasks, while the service pattern describes the distribution of services times of tasks. The arrival pattern to a queueing system is typically described in terms of the average time between two successive arrivals or the average number of arrivals per some unit of time. Third parameter describes the number of servers in a queueing system. The fourth parameter, the capacity, is the maximum number of customers allowed to enter the system. This quantity is often referred to as the buffer size and this can be either bounded or unbounded. Finally, the service discipline describes the manner in which the tasks are selected for service. First-come-first-served (FCFS), Last-come-first-served (LCLS) and shortest remaining processing time (SRPT) are a few examples of service disciplines, FCFS being the most common one.

2.1.2 Workload properties

The service time distribution (denoted by the second parameter of Kendall's notation) plays an important role when designing a task assignment policy. As was discussed earlier, this thesis proposes novel task assignment policies to assign tasks under heavy-tailed and arbitrary service time distributions. Under heavy-tailed distributions there is a very high probability that the size of task being

very small (short), while the probability that a size of task being very large (long) is very small. This results in a service time distribution that has a very high variance (second moment). Although the probability of very large task appearing is very small, the load imposed on the system by these (very small number of) large tasks can be as high as 50% of the system load. Moreover, when the service time distribution exhibits very high variance, several small tasks can get behind a very large task. This results in significant performance degradations, particularly if the tasks are processed in a FCFS manner until completion [Harchol-Balter, 2002; Broberg et al., 2006; Harchol-Balter et al., 1999; Kleinrock, 1975]. To what extent the performance degrades is determined by the variability of traffic (variability level) [Zikos and Karatza, 2010; Harchol-Balter, 2002; Crovella et al., 1998a]. Another important property of heavy-tailed service time distributions is the property of deceasing failure rate. This means that the longer a task has processed, the longer it is expected to continue processing.

It is important to realise that the term heavy-tailed is a generic term. When it comes to modelling and simulating performance of systems under such distributions, we need to carry out the analysis under a particular parametric heavy-tailed distribution. One of the most commonly used and widely appearing heavy-tailed distributions relating to computing workloads (e.g. Internet traffic) is the Pareto distribution. In this thesis we represent heavy-tailed traffic using the Bounded Pareto distribution. The Bounded Pareto distribution has been used extensively in previous work [Harchol-Balter, 2002; Broberg et al., 2004; 2006; Harchol-Balter et al., 1999; Tari et al., 2005; Fu and Tari, 2003; Crovella et al., 1998a] to represent heavy-tailed traffic, since it allows analytically tractable models and performance comparisons with numerous existing policies, which are also based on the Bounded Pareto distribution. The probability density function of the Bounded Pareto distribution is given by

$$f(x) = \frac{\alpha k^{\alpha}}{1-(\frac{k}{p})^{\alpha}} x^{-\alpha-1}, \qquad k \leq x \leq p, \tag{2.1}$$

where k, p and α represent the smallest task size, the largest task size and the variability of traffic respectively. α is called the tail parameter and it is inversely proportional to the variance of the service time distribution. The value of α depends on the type of tasks. For example, α lies in the range [1.1, 1.3] for the sizes of files transferred over the Internet [Crovella and Bestavros, 1997; Crovella et al., 1998b]. Unix process CPU requirements have an α value of 1.0 [Harchol-Balter and Downey, 1997]. α is typically computed off-line and the scheduling parameters (e.g. server cut-offs, fractions of tasks assigned to hosts, etc.) for policies are computed based on this α and certain other properties such as the average arrival rate.

In most cases the performance metrics (i.e. expected waiting or expected slowdown) of task assignment policies are associated with the moments of the service time distribution. The j^{th} moment

of the Bounded Pareto distribution is given by

$$E[X^j] = \begin{cases} \frac{\alpha k^\alpha (k^{j-\alpha} - p^{j-\alpha})}{(\alpha - j)(1-(k/p)^\alpha)}, & \text{if } \alpha \neq j, \\ \frac{k}{(1-(\frac{k}{p}))}(\ln p - \ln k), & \text{if } \alpha = j. \end{cases} \tag{2.2}$$

Throughout this thesis we assume that the upper bound, p (of the Bounded Pareto distribution) is fixed and is equal to 10^7. By letting p equal to a high value such as 10^7, we ensure that we represent realistic heavy-tailed service time distributions. It is also assumed that the mean (i.e. E[X]) of the Bounded Pareto distribution is equal to 3000 [Harchol-Balter et al., 1999].[1] These values have been used extensively in previous work when evaluating the performance of task assignment policies [Broberg et al., 2004; 2006; Harchol-Balter, 2002; Harchol-Balter et al., 1999].

2.1.3 Poisson process

The second important parameter in a queueing system is the arrival process. Many existing task assignment policies [Harchol-Balter, 2002; Broberg et al., 2004; 2006; Tari et al., 2005; Harchol-Balter et al., 1999; Zhang and Sun, 2005] are based on the assumption that tasks arrive at the system following a Poisson process. In this thesis we make the same assumption. This is a reasonable assumption under a wide range of traffic conditions, especially when we model the behaviour of aggregate traffic [Cao et al., 2001; Williamson, 2001].[2] Poisson process is a stochastic process (random variables indexed by time), where the probability of more than one task arriving at a given instance is equal to 0. When tasks are arriving according to a Poisson process, the number of tasks that arrive in two consecutive periods of time is independent of each other (independent increments). Moreover, when tasks arrive according to a Poisson process, it can be shown that the inter-arrival times follow an exponential distribution with the mean of $\frac{1}{\lambda}$, where λ is called the rate of the Poisson process.

The system load is defined as $\lambda E[X]$, where λ and $E[X]$ refer to the average arrival rate into the system and the mean of the service time distribution. In this thesis we evaluate the performance (of policies) under three different system loads where possible. They are 0.3 (a low system), 0.5 (a moderate system load) and 0.7 (a high system load).

[1] Note that the average service time of a web page is equal to 3000 bytes [Harchol-Balter et al., 1999].

[2] Poisson process typically appears in nature when we observe the aggregate effect of a large number of arrivals into systems.

2.1.4 Cluster-based distributed systems (server farms)

This thesis proposes task assignment policies for a homogeneous cluster (i.e. server farm) that consists of n number of back-end hosts that are identical to each other in all respects (i.e. processing, memory capacities, etc.). Figure 2.2 illustrates the host architecture of a typical server farm. This model consists of a front-end dispatching device (e.g. router, switch, etc.), which receives new tasks and directs these tasks to back-end hosts. The back-end hosts have the ability to broadcast or multicast requests to different back-end hosts. This architecture has become very popular in recent years due its cost effectiveness and scalability. A server farm can be constructed by networking a group of low cost commodity personal computers and capacity of the farm can be increased simply by adding more servers to farm.

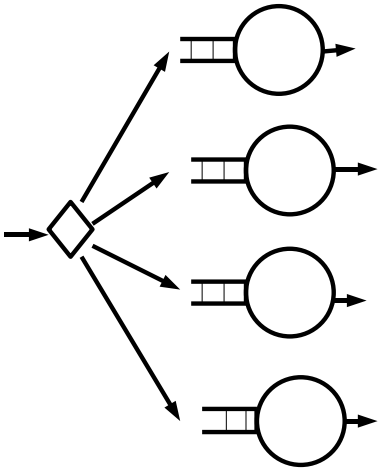


Figure 2.2: A cluster-based distributed computing system (Server farm)

There exist two types of server farms, namely, the batch computing server farms and the time sharing server farms. Batch computing server farms typically process tasks in a FCFS basis until completion or up to a predefined time limit. Under a batch computing system, if a task is preempted, then the preempted task is not generally processed at the same host rather processed at a different host.[3] Time sharing distributed systems (e.g. web server farms), on the hand, can preempt a currently processing task and process the task at a later stage (at the same host).

Certain server farms support task migration, whereas some others do not. Task migration can be of two types: work-conserving migration (preemptive migration) and non work-conserving migration

[3]Tasks processed in batch computing systems typically have specific requirements.

(non-preemptive migration). Under preemptive migration, information about current status of the task such as the process address space and the register content are migrated from the source host to the destination host, which resumes the execution of the task. Non-preemptive migration does not require the current status of the running tasks to be migrated from the source host to the destination node. Therefore, it is less expensive compared to that of preemptive migration. The drawback though is that this type of migration requires the task to be restarted from scratch at the destination host. In this thesis we consider both preemptive and non-preemptive task assignment policies.

2.1.5 Optimisation problems

Task assignment policies [Harchol-Balter, 2002; Broberg et al., 2004; 2006; Crovella et al., 1998a] (proposed for server farms) typically have scheduling parameters (e.g. server cut-offs, etc.) associated with them. In order to find the optimal values for these scheduling parameters, complex non-linear optimisation problems needed to be solved [Harchol-Balter, 2002; Broberg et al., 2004; 2006; Crovella et al., 1998a]. In this thesis we use the following two methods to solve these optimisation problems.

- Mathematica [Wolfram Research, 2003]: Mathematica is used in Chapter 3 to solve optimisation problems associated with single server systems.
- Particle swarm optimisation (PSO) [Kennedy and Eberhart, 1995] algorithm: Unfortunately, Mathematica cannot solve optimisation problems related to multi-server systems because these problems are extremely complex non-linear optimisation problems that have many decision variables. Therefore, in Chapters 4, 5 and 6, we use PSO to solve these problems. PSO is an evolutionary algorithm, which iteratively improves (optimises) its solution with respective to a given measure of quality. PSO places its particles in the search space of the objective function, where the objective function is evaluated at each iteration. The movement of the particles in the search space is determined by a simple Mathematical formula, which takes into account the position and the velocity of particles. It is not our intention to discuss the PSO in detail as here we simply use it as a technique for solving the optimisation problems. More details about PSO can be found in Kennedy and Eberhart [1995] and Poli et al. [2007].

Let us now consider a particular optimisation problem. This problem relates to a scheduling policy called Multi-level-time-sharing (MLTP) which is discussed in Chapter 3. In this optimisation problem quanta for MLTP is computed such that the expected waiting time is minimised. Figure 2.3 compares the (minimum) expected waiting time (denoted by $E[W]$) obtained using Mathematica and PSO. We

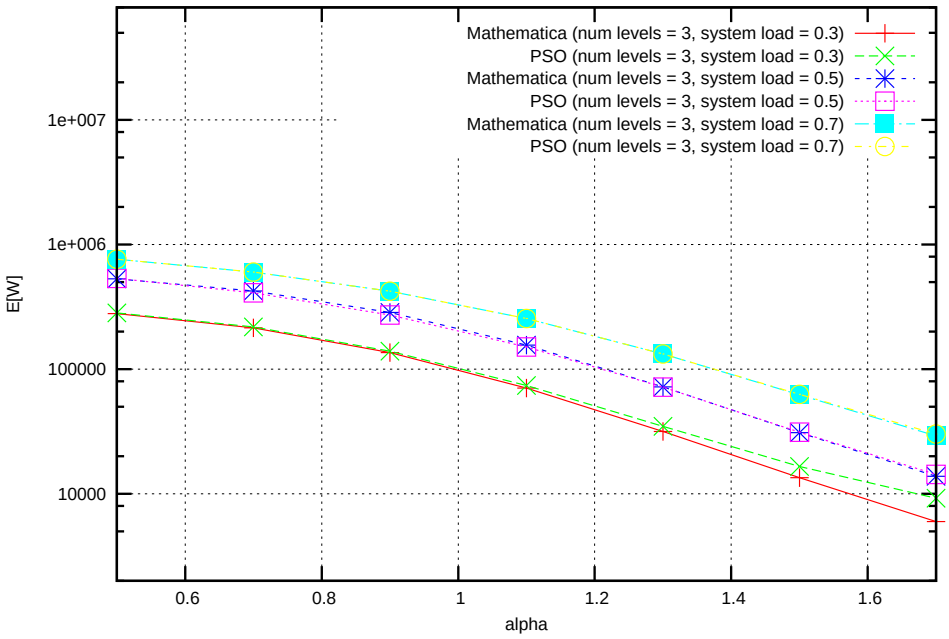


Figure 2.3: Evaluation of PSO using MLTP

note that the results obtained using Mathematica and PSO algorithm are more or less the same for all the scenarios.

2.2 Scheduling policies

Scheduling is the fundamental way to minimise the response times of tasks, where even a small change to the scheduling policy can result in a massive improvement in the performance. This section discusses some basic concepts of scheduling. Scheduling policies can be categorised into two main types: preemptive scheduling policies and non-preemptive scheduling policies. The difference between preemptive and non-preemptive scheduling policies is that non-preemptive scheduling policies process tasks until completion (without interruption), whereas preemptive scheduling policies can suspend a currently executing task and resume its execution at a later stage.

2.2.1 Non-preemptive scheduling policies

This section discusses four important non-preemptive scheduling policies: 1) First-come-first-served (FCFS), 2) Last-come-last-served (LCFS), 3) Random and 4) Shortest Job First (SJF).

- **First-come-first-served (FCFS):** FCFS [Kleinrock, 1975; Gross and Harris, 1998] is the most commonly used and widely appearing scheduling policy, which processes tasks on a first come first served basis until completion. When the system finishes processing a task, the task at the head of queue will be served next.

- **Last-come-first-served (LCFS):** When the server finishes processing a task, LCFS [Kleinrock, 1975; Gross and Harris, 1998] processes the task that arrived last at the system (until completion).

- **Random:** Random [Gross and Harris, 1998; Kleinrock, 1975] chooses the next task to be served randomly and processes it until completion.

  It is important to point out that the expected waiting time for FCFS, LCLS and Random are the same. The expected waiting time for $M/G/1$ FCFS queue is given by the Pollaczek-Khinchin (P-K) formula [Kleinrock, 1975]

  $$E[W]_{FCFS} = \frac{\lambda E[X^2]}{2(1-\lambda E[X])}, \tag{2.3}$$

  where $E[X]$ and $E[X^2]$ denote 1^{st} and 2^{nd} moments of the service time distribution of tasks. λ denotes the average arrival rate into the system.

  Note that the variances of the waiting time under these policies, however, are not the same. Let $Var(T)_{(FCFS)}$, $Var(T)_{(Random)}$ and $Var(T)_{(LCLS)}$ be the variance of waiting time under FCFS, LCLS and Random. Then,

  $$Var(T)_{(FCFS)} < Var(T)_{(Random)} < Var(T)_{(LCLS)}. \tag{2.4}$$

  The variance of waiting time under LCLS is very high compared to that of FCFS.

- **Shortest Job First (SJF):** Under SJF [Kleinrock, 1976], when the system finishes processing a task, the task with the smallest size is served next. While this task is being processed, if the system receives a new task that has a smaller task size (than the current task) then the new task will only be served after processing the current task. SJF outperforms FCFS under a wide range of workload conditions. However, SJF could significantly penalise large tasks under certain workload conditions. In addition, it assumes that the sizes of tasks are known in advance. This means that it cannot be used for scheduling tasks such as dynamic web content whose service times are not known in advance.

- **Non-preemptive priority scheduling:** A non-preemptive priority scheduling system [Kleinrock, 1975; 1976] maintains a separate queue for each priority class. When the system finishes processing a task, the task at the end of (non-empty) highest priority queue is processed next. While this task is being processed, if the system receives a new task that has a higher priority (than the one being processed), then the new task will only be served after processing the current task (until completion).

2.2.2 Preemptive scheduling policies

This section discusses preemptive scheduling policies. These policies are used in various time sharing systems (such as server farms and routers) to schedule tasks.

- **Round-Robin:** Under Round-Robin [Kleinrock, 1975; 1976] tasks are served in a FCFS manner up to a maximum amount of time called the quantum. If the service time of the task is less than or equal to the quantum, then the task departs the system. Otherwise, the task is preempted and placed at the end of the queue, where the task is served in a similar manner (up to the quantum). This process continuous until the task is fully processed at which point the task departs the system.

- **Processor Sharing (PS):** The limiting case of Round-Robin when the quantum size approaches zero is called the processor sharing. The expected waiting time for M/G/1 PS queue [Kleinrock, 1975; 1976] is given by

$$E[W] = \frac{\lambda E[X]}{1 - \lambda E[X]}, \tag{2.5}$$

  where λ and $E[X]$ represent the average arrival rate into the system and the mean of the service time distribution respectively. M/G/1 PS performs better than M/G/1 FCFS, if the squared coefficient of variance (SCV) of the service time distribution is greater than 1. We note that the expected waiting time for PS does not depend on $E[X^2]$, the second moment of the service time distribution, which indicates that the variance of the service time distribution has no impact on the expected waiting time. This has to do with the fact that PS assumes a infinitely small quantum. In real systems the quantum is not infinitely small (never zero) and it is a positive value, which is computed based on a number of factors (such as the context switch overhead).

- **Foreground-background scheduling (FB)**: FB [Nuyens and Wierman, 2008] is a preemptive scheduling policy, which gives priority to the task with the least amount of service. Let us briefly explain the behaviour of FB using two tasks. Let us assume that Task 2 arrives at the

system, while Task 1 is being serviced. When Task 2 arrives at the system, Task 1 has received certain amount of service. Let this be x. Upon the arrival of Task 2, Task 1 is preempted from service and Task 2 is serviced up to x. Once it receives x amount of service, both tasks are serviced concurrently until completion. The expected waiting time of a task of size x and the overall expected waiting time of a task under M/G/1 FB are given by

$$E[W(x)] = \frac{x}{1-\lambda \int_0^x \overline{F(t)}dt} + \frac{\lambda \int_0^x t\overline{F(t)}dt}{(1-\lambda \int_0^x \overline{F(t)}dt)^2}, \tag{2.6}$$

$$E[W] = \int_0^\infty E[W(x)]f(x)dx, \tag{2.7}$$

where $F(t)$ and λ denote the service time distribution and the average arrival rate into the system respectively. $\overline{F(t)} = 1 - F(t)$ is the survival function.

Let $E[W]^{FB}$ and $E[W]^{PS}$ be the expected waiting time for FB and expected waiting time for processor sharing respectively. If the service time distribution of tasks has a decreasing failure rate (DFR, i.e. the longer a task has run, the longer it is expected to continue to run), then

$$E[W]^{FB} < E[W]^{PS}. \tag{2.8}$$

If the service time distribution has an increasing failure rate, then

$$E[W]^{FB} > E[W]^{PS}. \tag{2.9}$$

If the service time distribution has a constant failure rate (i.e. exponential distribution), then

$$E[W]^{FB} = E[W]^{PS}. \tag{2.10}$$

FB is not efficient because it preempts the current task whenever a new task arrives at the system. Under high arrival rates, this can be very inefficient and costly. Furthermore, FB can be difficult to implement in time sharing systems because time sharing systems process tasks up to a fixed amount of time (quantum). If the size of a task is greater than the quantum, the task is preempted (from service) and processed at a later stage.

- **Multi-level time sharing (MLTP):** MLTP [Schrage, 1967] has recently gained considerable attention because it has shown improved performance under modern (realistic) traffic condi-

tions (e.g. service time distributions with the property of decreasing failure rate, service time distributions with high variance, etc.). MLTP consists of multiple queues and it gives preferential treatment to tasks with small service times. MLTP is discussed in detail in Chapter 3.

- **Multi-level processor sharing (MLPS):** MLPS [Schrage, 1967; Aalto et al., 2007; 2005] is the limiting case of MLTP, where the quantum approaches zero and the number of levels approaches infinity. Under specific conditions, FB can be considered a coarse-grained approximation of MLPS.

2.3 Traditional task assignment policies

Existing task assignment policies proposed for assigning tasks in server farms can be categorised into two main types, namely, the traditional task assignment policies and advanced task assignment policies. This section presents the details of four well known traditional task assignment policies, namely, Random, Round-Robin, Central-Queue and Join the Shortest Queue. These policies only perform well under exponential service time distributions and their performance is very poor under empirical workloads (such as heavy-tailed service time distributions). However, they are still widely being used due to their simplicity, despite their major drawbacks, which include their poor performance under fluctuating workload conditions, poor performance under non-exponential service time distributions and inability to support task migration between hosts. Let us now discuss these policies.

- **Random:** Under Random [Silberschatz et al., 1998], each task arriving at the central dispatcher is assigned to a back-end with an equal probability. Back-end hosts process tasks according to a particular scheduling policy until completion. Waiting time analysis for Random is rather straightforward. Note that the Random scheduling policy (discussed in Section 2.2) is different from the Random task assignment policy.

  Let us now consider a particular example, where the back-end hosts of Random policy process tasks according to FCFS policy until completion. Let $E[W_i]$ and p_i be the expected waiting time of a task at Host i and the probability that a task is dispatched to Host i respectively. Then, the expected waiting time of a task in the system, $E[W]$ is given by

$$E[W] = p_1E[W_1] + p_2E[W_2] + ... + p_iE[W_i] + ... + p_nE[W_n], \tag{2.11}$$

  where n denotes the number of hosts in the farm.

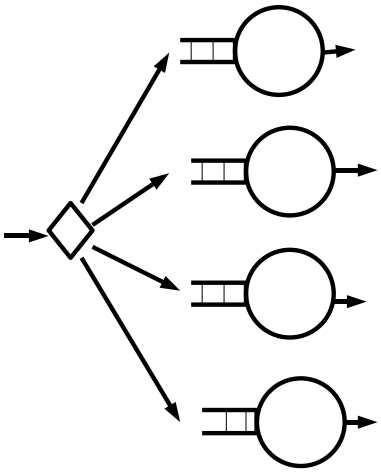


Figure 2.4: Random task assignment policy

Since, Random dispatches task with an equal probability

$$p_1 = p_2 = ... = p_n = \frac{1}{n}. \tag{2.12}$$

$E[W_i]$ is obtained using the Pollaczek-Khinchin formula [Kleinrock, 1975]

$$E[W_i] = \frac{\lambda_i E[X^2]}{2(1-\lambda_i E[X])}, \tag{2.13}$$

where $E[X]$ and $E[X^2]$ represent 1^{st} and 2^{nd} moments of the service time distribution respectively and λ_i denotes the average arrival rate into Host i.[4] The quantity $\lambda_i E[X]$ is typically referred to as the system load and it is denoted by ρ_i.

Since, Random dispatches tasks with an equal probability

$$\lambda_1 = \lambda_2 = ... = \lambda_n = \frac{\lambda}{n}, \tag{2.14}$$

where λ represents the average arrival rate of tasks at the dispatcher. Hence, we get

$$E[W] = E[W_1] = E[W_2] = ... = E[W_n]. \tag{2.15}$$

[4]Note that under Random, each host in the system sees the same processing time distribution.

Figure 2.5 shows the expected waiting for Random under 3 different system loads. These results have been obtained for a Bounded Pareto service time distribution under the conditions discussed in Section 2.1.2. We note that the expected waiting time for Random (when tasks are

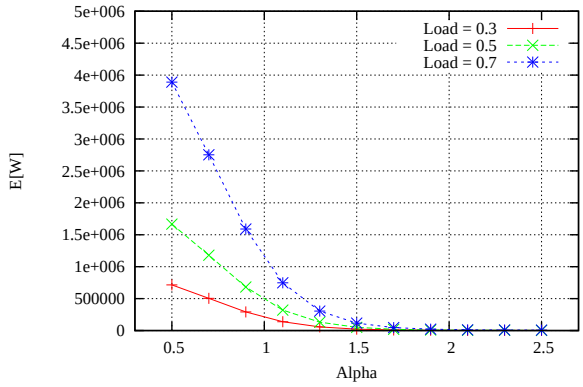


Figure 2.5: Expected waiting time for Random

processed in FCFS manner with no preemption) depends on two factors, namely, the system load and α, where α represents variability of task sizes. When α is fixed, the expected waiting time increases with the system load. For example, under a system load of 0.3, when α equals 1.5, the expected waiting time for Random is equal to 21537, while under a system load of 0.7, when α equals 1.5, the expected waiting is equal to 117257.

As the variability of traffic increases (i.e. as α decreases), the expected waiting time for Random increases rapidly. We note that under very high task size variabilities (e.g. α = 0.5), the performance of Random is extremely poor. For example, under a system load of 0.7, when $\alpha = 0.5$, the expected waiting time for Random is computed as 505477, while under the same system load, when $\alpha = 2.5$, the expected waiting time is computed as 1142. As we note from Equation 6.17, the expected waiting time is proportional to the second moment of the service time distribution, which is very high when α is low.

- **Round-Robin:** Under Round-Robin [Silberschatz et al., 1998], tasks are assigned to back-end hosts in a cyclical fashion and are processed at hosts using a particular scheduling policy (e.g. FCFS) until completion. Both Random and Round-Robin attempt to equalise the expected number of tasks at servers. The performance of Round-Robin is very similar to Random and this has been discussed in [Harchol-Balter, 2002]. There exist two variants of Random and Round-Robin, called Weighted-Random and Weighted-Round-Robin respectively, which have

been designed for heterogeneous server farms, where the servers have varying degrees of processing power. These policies assign higher number of tasks to hosts with higher processing capacities. Note that the Round-Robin scheduling policy (discussed in Section 2.2) is different from the Round-Robin task assignment policy.

- **Central-Queue (CQ):** The dispatcher of the Central-Queue policy [Weber, 1978] holds newly arriving tasks in a FCFS queue until a host (in the farm) is idle. Once a host becomes idle, the task at the head of the queue is assigned to that host. The task is processed in a FCFS manner until completion at that host. Central-Queue performs better compared to Random and Round-Robin under exponential service time distributions [Weber, 1978]. However, it does not perform well under other service time distributions (e.g. Pareto).

- **Join the Shortest Queue (JSQ)**: Under JSQ [Winston, 1977], the central dispatcher assigns incoming tasks to the back-end host that has the least number of tasks in its queue and the tasks are (generally) processed in a FCFS manner at hosts until completion. Under exponential service time distributions, JSQ has shown better performance compared to other traditional task assignment policies [Winston, 1977].

2.4 Advanced task assignment policies

As pointed out traditional task assignment policies possess numerous drawbacks. Therefore, to address these problems numerous advanced task assignment policies have been proposed in recent years. This section provides the details of some major contributions made in recent years. These policies have been proposed for different types of environments and therefore, they are based on different types of assumptions. Some desirable properties of a good task assignment policy are

- Ability to efficiently assign tasks in time sharing systems - Time Sharing Systems.
- Ability efficiently assign tasks in batch computing systems - Batch Computing Systems.
- Ability to efficiently assign tasks with unknowns sizes - Unknown Task Sizes.
- Ability to efficiently assign tasks under heavy-tailed service time distributions - Heavy-tailed Service Time Distributions.
- Ability to efficiently assign tasks with unknown (arbitrary) service time distributions - Arbitrary Service Time Distributions.

- Ability to efficiently assign tasks under changing workload conditions - Dynamic Task Assignment.
- Ability to efficiently assign tasks in multiple server farm environments - Multiple Server Farms.
- Ability to migrate tasks between hosts without restarting the tasks at destination hosts - Preemptive Migration.

In practice it is not possible to design a task assignment policy to have all of the above features. However, when we design a new task assignment policy, it is important to take into account the above factors together with specific system properties and traffic properties. The following (advanced) task assignment policies are discussed in this chapter.

- Task Assignment based on Guessing Size (TAGS) [Harchol-Balter, 2002]
- Task Assignment based on Guessing Size with Preemptive Migration (TAGS-PM). This task assignment policy is also called Task Assignment based on Guessing Size with Work-conserving Migration (TAGS-WC) [Harchol-Balter, 2002; Broberg et al., 2006]
- Task Assignment based on Prioritising the Traffic Follow (TAPTF) [Broberg et al., 2004]
- Task Assignment with Work-conserving Migration (TAPTF-WC) [Broberg et al., 2006]
- Size Interval Task Assignment with Equal Load (SITA-E) [Harchol-Balter et al., 1999]
- Size Interval Task Assignment with Variable Load (SITA-V) [Crovella et al., 1998a]
- Least-loaded Server First (LLF) [Tari et al., 2005]
- A Least Flow-Time First Load Sharing (LFF-SIZE) [Tari et al., 2005]
- ADAPTLOAD [Zhang and Sun, 2005]
- EQUILOAD [Ciardo et al., 2001]

The reason for concentrating on the above policies is because they perform well under certain empirical workload conditions (e.g. heavy-tailed) and to improve the performance under such conditions, they use special techniques (such as unbalancing the load among hosts, reducing the task size variability in hosts queues, multi-section queues, task migration, etc.). Performance analysis of these policies indicates that they perform significantly better than traditional policies. For example, TAGS outperforms Random by a factor of 100 under highly variable traffic conditions. Although these

policies perform well in certain environments, they possess several major limitations. For example, SITA-V, SITA-E, LFF-SIZE, ADAPTLOAD and EQUILOAD assume that the service times of tasks are known in advance. Therefore, these policies cannot be used for assigning dynamic content and scientific workloads, whose service times cannot be estimated prior to execution. Moreover, none of the above policies can effectively assign tasks in multiple server farms, nor can they efficiently assign tasks when the service time distribution of tasks is unknown. Furthermore, most of these policies have been designed for assigning tasks in batch computing server farms. As such, they are not suitable for assigning tasks in time sharing farms. The following table illustrates the properties of each of the above task assignment policies.

Table 2.1: A list of desirable properties for task assignment policies

Property	TAGS	TAGS-PM	TAPTF	TAPTF-WC	SITA-E	LFF-SIZE	ADAPT LOAD	EQUI LOAD
Time Sharing Systems	-	-	-	-	-	-	√	-
Batch Computing Systems	√	√	√	√	√	√	-	√
Unknown Task Sizes	√	√	√	√	-	-	-	-
Heavy-tailed Service Time Distributions	√	√	√	√	√	√	√	-
Arbitrary Service Time Distributions	-	-	-	-	-	-	-	-
Dynamic Task Assignment	-	-	-	-	-	√	√	-
Multiple Server Farms	-	-	-	-	-	-	-	-
Preemptive Migration		√	-	√	-	-	-	-

2.4.1 Task Assignment based on Guessing Size (TAGS)

TAGS [Harchol-Balter, 2002; 2000] is a well known policy, which has been proposed for assigning tasks in batch computing server farms. TAGS policy is illustrated in Figure 2.6. Throughout this thesis we use TAGS as a baseline for our performance evaluations. TAGS assumes a homogeneous web server farm that consists of n number back-end hosts and a central dispatcher. The main functionality of TAGS is as follows. The central dispatcher assigns each new task to Host 1, which processes tasks

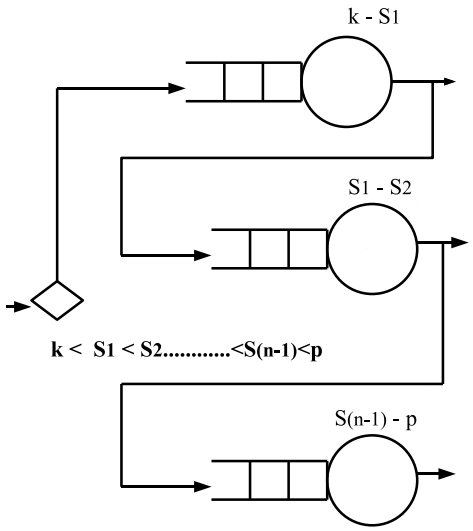


Figure 2.6: TAGS task assignment policy

in a FCFS manner up to a predefined processing time limit (assigned to Host 1). If the service time of the task is less or equal to the processing time limit assigned to Host 1, the task departs the system (after receiving its full amount of service) and the results are sent back to the client. Otherwise, the task is migrated to the next host. The next host processes the task from scratch up to its predefined time limit and so on. This process continues until the task is fully processed at which point the task departs the system. The performance under TAGS depends on the variability of traffic, average arrival rate and task size ranges (i.e. cut-offs). Let s_i denotes the upper limit of the task size range associated with Host i. Let k and p represent the smallest and the largest task sizes in the service time distribution. The size ranges for hosts are computed to optimise a certain performance metric under a given α and system load such that

$$k < s_1 < s_2 ... < s_{i-1} < s_i < ... < s_n = p \tag{2.16}$$

TAGS has been especially designed to assign tasks that exhibit high variance in their task sizes and therefore, it has shown significant performance improvements over policies such as Random, Round-Robin and Central-Queue under a wide range of such workload conditions. However, TAGS performs poorly under specific workload conditions. Let us now investigate the performance of TAGS in detail.

(a) Expected waiting time for TAGS: 2 Host case

Here we investigate the expected waiting time for TAGS in a 2 Host system. Figure 2.7 illustrates the expected waiting time for TAGS under 3 different system loads.[5] We note that TAGS performs

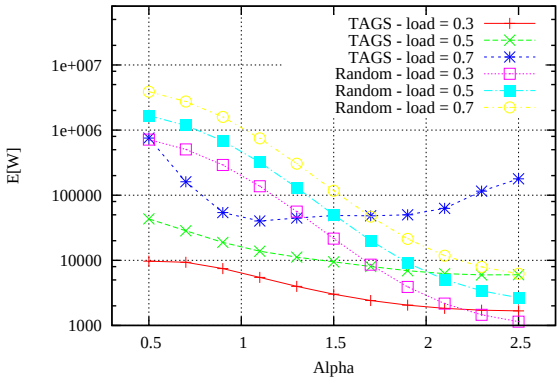


Figure 2.7: Expected waiting time for TAGS in a 2 Host system

exceptionally well compared to Random under a range of scenarios. Under low and moderate system loads, the expected waiting time for both Random and TAGS improves with increasing α. This is because both policies process tasks in a FCFS manner at each host. As was pointed out, the expected waiting time of a task in a FCFS queue is directly proportional to variance of tasks sizes. As α increases, the variance of task sizes in each host queue decreases leading to an improvement in the overall performance.

Under a system load of 0.7, the expected waiting time for TAGS decreases up to a certain value and then it begins to increase. This behaviour of TAGS has not been discussed in [Harchol-Balter, 2002; 2000]. The reason for this behaviour is as follows. Low α parameter indicates that there are a very small number of tasks with very long processing requirements that constitutes to a large fraction of the total workload. Under low α values, this small fraction of very large tasks are processed until completion at Host 2. As α increases, the fraction of tasks with very long processing requirements (that make up a large fraction of the workload) decreases. As a result, more tasks are migrated to Host 2 in order to ensure that Host 1 does not get overloaded. This leads to an increase in the excess load, particularly under high system loads (e.g. 0.7), which leads to poor performance.

(b) Expected waiting time for TAGS: 3 Host case

[5]Note that in Figure 2.7 we have used log scale for y axis.

Figure 2.8 plots the expected waiting time for TAGS and Random in a 3 Host system. We note that

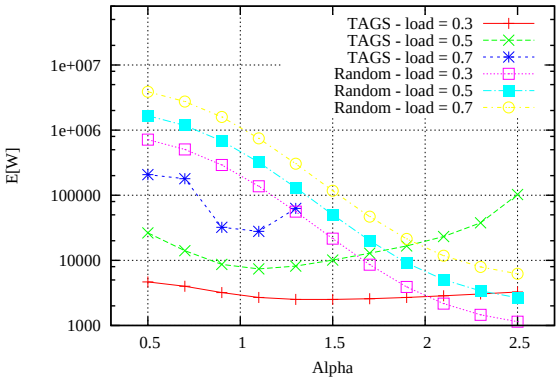


Figure 2.8: Expected waiting for TAGS in a 3 Host system

TAGS outperforms Random significantly if α is low. For example, under a system load of 0.5, when α equals 0.5, TAGS outperforms Random by a factor of 63. However, TAGS has very poor performance under very low task size variabilities and very high system loads. For example, under a system load of 0.5, when $\alpha = 2.5$, Random outperforms TAGS by a factor of 38. The reason for this poor performance (of TAGS) is the very high excess load generated under these conditions.

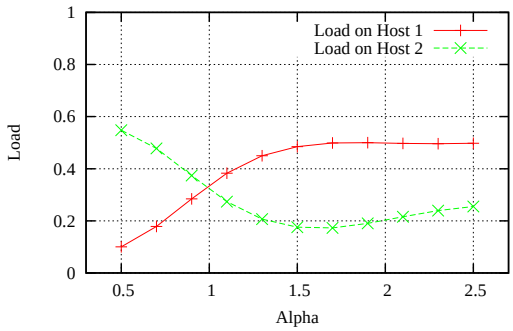
(c) Load on hosts

We noted that TAGS can result in significant performance improvements under a wide range of workload conditions. Here we investigate the behaviour of load on individual hosts. We note that TAGS improves the performance by unbalancing the load among hosts. Figure 2.9 plots the load on individual hosts in a 2 Host systems. We note that when α is low, the load on Host 2 is significantly higher than the load on Host 1. For example, under a system load of 0.5, when α equals 0.5, the load on Host 1 and Host 2 are equal to 0.25 and 0.87 respectively.[6] As α increases, the load on Host 1 increases, whereas the load on Host 2 decreases. It is interesting to note that when α is near 1, the load on hosts are more or less the same under all three system loads considered.

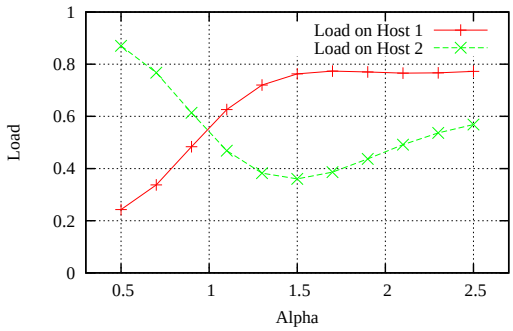
2.4.2 Task Assignment based on Guessing Size with Preemptive Migration (TAGS-PM)

The original version of TAGS discussed in the previous section restarts certain tasks from scratch. This results in an additional load on the system, which we called the excess load. The total excess

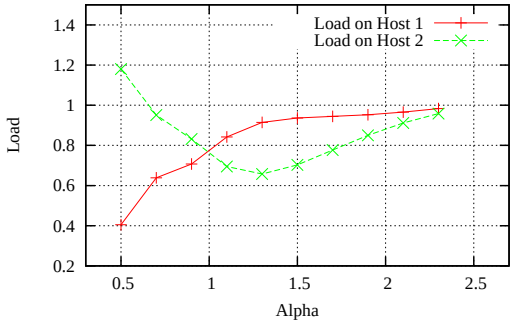
[6]Note that the average of the sum these two loads is greater than the system load because system load does not take into account the excess load.



(a) System load = 0.3



(b) System load = 0.5



(c) System load = 0.7

Figure 2.9: Load on individual hosts under TAGS in a 2 Host system

load is dependent on a number of factors, which include the variability of traffic, arrival rate and number of hosts in the system. TAGS-PM is based on preemptive (work-conserving) task migration and this means that it does not restart tasks from scratch, rather it resumes the execution of tasks at destination hosts. The core functionality of TAGS-PM is similar to TAGS except that TAGS-PM resumes the execution of tasks rather than restarting them from scratch. Figures 2.10 and 2.11 plot the expected waiting time vs α for TAGS-PM and TAGS in 2 and 3 Host systems respectively.

Let us first consider the expected waiting time for the two policies in a 2 Host system. We note that there is a significant difference between the behaviour of expected waiting time under TAGS and TAGS-PM, particularly under a system load of 0.7. Under a system load of 0.7, the expected waiting time for TAGS-PM decreases with α up to certain value and then it increases slightly and then it deceases again. This is different from TAGS, where the expected waiting time deceases up to a certain limit and it then continuously increases with α. We note that when α is in the proximity of 2.5, TAGS has very poor performance compared to Random. TAGS-PM, on the other hand, outperforms

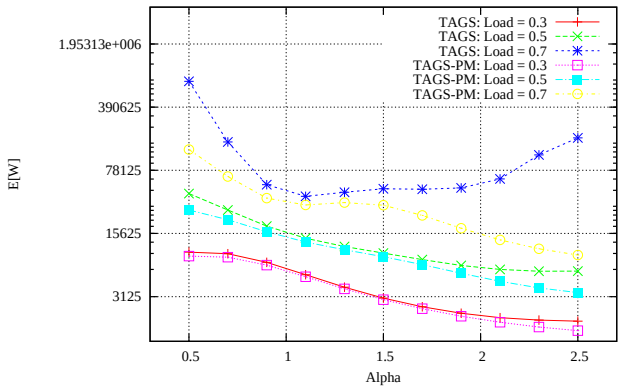


Figure 2.10: Expected waiting time for TAGS and TAGS-PM in a 2 Host system

Random under both low and high α values. For example, under a system load of 0.7, when α is equal to 0.5, it outperforms Random by a factor of 30. Under the same system load, when α equals 1.9, TAGS-PM outperforms Random by a factor 1.1. Performance under TAGS-PM does not deteriorate with increasing α, since it does not restart tasks from scratch and as such it does not generate any excess load in the system.

We note that the behaviour of expected waiting time for TAGS-PM in a 3 Host system is similar to that of a 2 Host system. Unlike TAGS, the expected waiting time does not degrade under high system loads, when α is in the proximity of 2.5. Moreover, we note that TAGS-PM having better performance in a 2 host system compared to that of 3 host system under all the scenarios considered. Therefore, TAGS-PM scales well. TAGS, on the other hand, does not scale well, particularly under high system loads. TAGS-PM, however, has one major problem that is it does no take into account the migration cost, which can be significantly high for particular types of tasks and systems.

We noted that for TAGS the load on hosts vary depending on the α value and system load. Similar observations are made with regard to the load on TAGS-PM. Figure 2.12 shows the behaviour of load for TAGS-PM in a 2 Host systems. Clearly, TAGS-PM improves performance by unbalancing the load among hosts.

2.4.3 Task Assignment based on Prioritising the Traffic Follow (TAPTF) and Task Assignment with Work-conserving Migration (TAPTF-WC)

This section presents the details of two other recent advanced task assignment policies, namely, TAPTF [Broberg et al., 2004] and TAPTF-WC [Broberg et al., 2006]. Both of these policies have

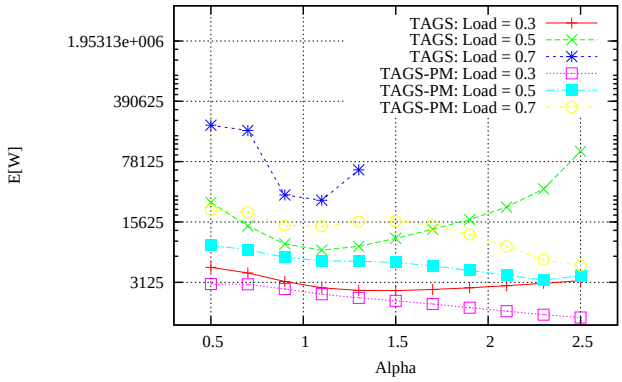
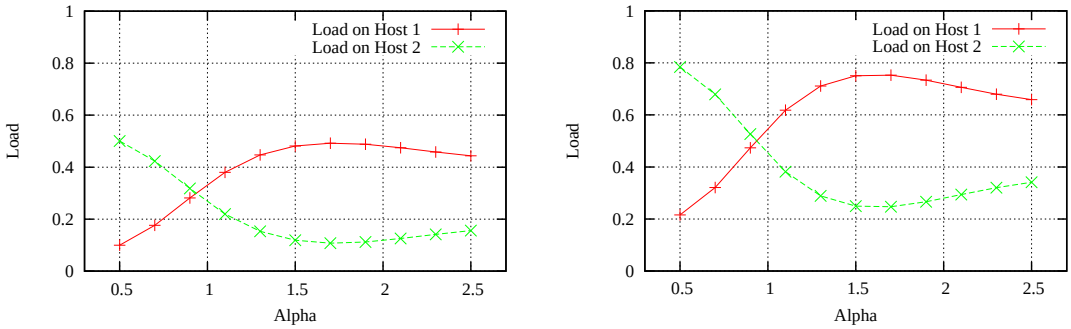
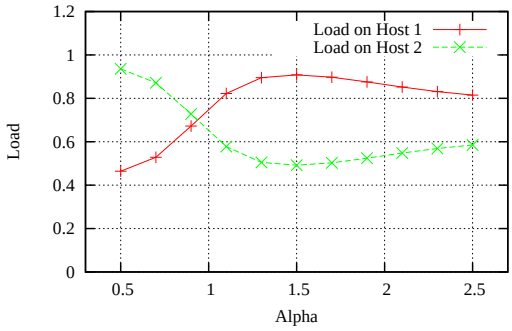


Figure 2.11: Expected waiting time for TAGS and TAGS-PM in a 3 Host system



(a) System load = 0.3

(b) System load = 0.5



(c) System load = 0.7

Figure 2.12: Load on individual hosts under TAGS-PM in a 2 Host system

been proposed for assigning tasks in batch computing environments. Let us first consider TAPTF policy. TAPTF has been proposed to address a major issue associated with TAGS, which is the high excess load (generated under certain workload conditions), which results in significant performance degradations. TAPTF attempts to minimise the excess load (due to restarting tasks from scratch) via its novel queueing architecture, which utilises two queues for each host. TATPF policy is depicted in Figure 2.13. We see Figure from 2.13 that each host in TAPTF (except Host 1) has two queues. These

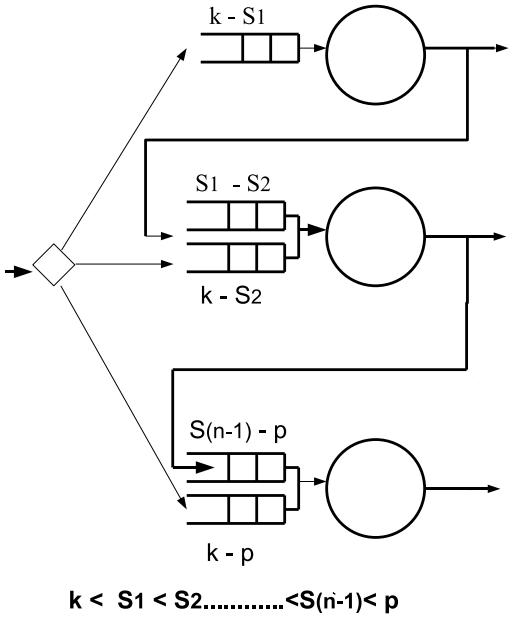


Figure 2.13: TAPTF task assignment policy

queues are called the ordinary queue (O queue) and the restart queue (R queue) respectively. The task in the O queue has the priority of service over tasks in R queue. Similar to TAGS and TAGS-PM, TAPTF has task size ranges associated with the hosts. Host i's O queue deals with tasks whose sizes are in the range $[k, s_i]$, while Host i's R queue deals with tasks whose sizes are in the range $[s_{i-1}, s_i]$.

The basic functionality of TATPF is follows. New tasks arrive at the central dispatcher following a Poisson process. The central dispatcher assigns these tasks to one of the back-end hosts with probability q_i, where q_i is the probability of a task being assigned to Host i. These tasks are placed in the ordinary queue (O queue) and processed up to s_i in a FCFS manner, where s_i is the upper limit of the task size range assigned to Host i. If the size of a task is less than or equal to s_i, the task departs the system. Otherwise, the task is placed in the R queue of the next host, where the task is restarted

from scratch. This process continuous until the task is fully serviced at which point the task departs the system. Performance analysis of TAPTF shows that TAPTF outperforms TAGS under certain α values. However, as α decreases (i.e. the variance of task sizes increases), TAGS starts to outperform TAPTF. Therefore, which policy performs better is determined by the variance of the service time distribution. Another issue with TAPTF is that it has a rather complex queueing model, which may be difficult to implement in real computing systems.

There is only one difference between TAPTF-WC and TAPTF, i.e. TAPTF-WC supports work-conversing migration. Similar to TAGS-PM, TAPTF-WC is based on the assumption that migrating tasks between hosts incurs no additional load on destination hosts or source hosts. As such, it generates no excess load on the system. Analytical performance comparison of TAPTF-WC with TAGS-PM shows that TAPTF-WC outperforms TAGS-PM under moderate α values, whereas TAGS-PM outperforms TAPTF-WC under low α values.

2.4.4 Size Interval Task Assignment with Equal Load (SITA-E) and Size Interval Task Assignment with Variable Load (SITA-V)

This section presents the details of two well known task assignment policies called SITA-E and SITA-V. These policies have been designed for server farms that serve static web content whose service times can be estimated prior to execution. These polices have their own way of addressing the problem of how to improve the performance under various workload scenarios, which is a much simpler problem when the service times of tasks are known in advance. When the service times of tasks are known *a priori*,

- it is possible to compute the load on servers simply by considering sizes of tasks in server queues. Such computations will allow tasks to be placed at the server with the least load,
- the dispatcher can assign each incoming tasks to the server with the correct size range, where each task is processed until completion,
- it is not required to design complex queueing models in order to optimise the performance,
- there is no need to solve complex optimisation problems to find the task size ranges for servers,
- there is no need to migrate tasks between hosts because tasks are always dispatched to the correct host, which processes tasks until completion,
- it is relatively easy to design adaptive task assignment policies, which can assign tasks under changing operating conditions.

Both SITA-E and SITA-V are size-based task assignment policies, which compute the task size ranges for hosts based on the service time distribution of tasks. SITA-E computes the size ranges for hosts such that the expected load at each host is the same, whereas SITA-V computes these size ranges (in an increasing order) so as to optimise a specific performance metric (e.g. expected waiting time). The central dispatchers of SITA-E and SITA-V assign incoming tasks to hosts based on their size, where tasks are processed in a FCFS manner until completion. The main purpose of having different size ranges for hosts is to speed up the flow of small tasks. This ensures that the small tasks do not get behind large tasks. This results in an improvement in the overall performance of the system under highly variable workloads.

Experimental and analytical performance analysis of SITA-E shows that it significantly outperforms traditional policies such as Random and Round-Robin. Moreover, it outperforms LLF (refer to Section 2.4.5) under specific workload scenarios (e.g. high workload variabilities). SITA-V, on the other hand, outperforms SITA-E under a wide range of workload scenarios including both high and low task size variabilities. The reason why SITA-V performs significantly better than SITA-E is because SITA-V unbalances the load among its hosts as opposed to balancing the load. In sections 2.4.1 and 2.4.2 we noted that both TAGS and TAGS-PM also use the same technique for improving the performance.

2.4.5 Least-loaded Server First (LLF) and Least Flow-time First Load Sharing (LFF-SIZE)

LLF (sometimes called Dynamic) dynamically assigns tasks to the server with the least load. The least loaded server is the server with the least amount of remaining work. The least loaded server is determined by considering the sizes of tasks (already) in host queues and the remaining processing time of the task currently being processed. One of the main limitations of LLF is the high computational overhead associated with computing the least loaded server. The performance comparisons of LLF with traditional policies indicate that LLF performs significantly better than traditional policies under a range of workload conditions. SITA-E, however, outperforms LLF under a wide range of scenarios, particularly under high task size variabilities.

LFF-SIZE is an improved version of LLF. LFF-SIZE utilises multi-section queues at its hosts to reduce the size variance of tasks. These multi-section queues accommodate tasks with different sizes. When the tasks are processed at hosts, small tasks are processed prior to large tasks with no preemption. When assigning tasks to hosts, tasks are assigned to the fittest server, which is determined based on the remaining work of individual hosts and processing capacities of servers. Performance analysis of LFF-SIZE shows that it outperforms LLF under a wide range of scenarios. Moreover,

it outperforms SITA-E under moderate task size variabilities. The performance of LFF-SIZE is not clear under high task size variabilities as the performance of LFF-SIZE has not been evaluated under these conditions. It is also not clear how well LFF-SIZE performs compared to SITA-V because the performance of LFF-SIZE has not been compared with SITA-V.

2.4.6 EQUILOAD and ADAPTLOAD

Similar to SITA-E, the hosts of EQUILOAD [Ciardo et al., 2001] have specific size ranges associated with them. The main difference between the two policies is that EQUILOAD uses a novel probabilistic method to compute boundaries of these ranges using the empirical data sets. This method involves estimating the probability density and cumulative distribution functions of the service times by fitting the empirical data into phase-type distributions (off-line). The performance analysis of EQUILOAD shows that it outperforms SRPT and JSQ under high task size variabilities or high system loads. The performance of EQUILOAD is similar to SITA-E apart from the fact that to compute size ranges (boundaries) EQUILOAD uses empirical data sets, while SITA-E uses synthetic workload distributions. Finally, it is important to note that EQUILOAD cannot adjust its boundaries online. The authors are considering this in their future work.

ADAPTLOAD [Zhang and Sun, 2005] is also a size-based task assignment policy, which has task size ranges associated with its hosts. It can dynamically compute the size ranges for hosts according to the changes that occur in the incoming traffic. These size ranges are computed using the discrete histogram of service times, which is estimated on-line. Performance analysis of ADAPTLOAD shows that it outperforms JSQ under changing arrival rates and non-stationary service time distributions.

2.5 Conclusion

The detailed discussion of existing task assignment policies provided in this chapter further elucidates the major limitations of existing task assignment policies, which we have already discussed in Chapter 1. In a nutshell traditional task assignment policies are not well suited for assigning tasks in server farms because their performance is extremely poor under realistic workload scenarios. On the other hand, advanced tasks policies possess some major drawbacks. For example, many advanced policies such as SITA-E, SITA-V, LFF-SIZE, EQUILOAD and ADAPTLOAD are based on the assumption that the service times of tasks are known *a priori*. Those that do not make any assumptions regarding the actual service times of tasks (e.g. TAGS, TAGS-PM, TAPTF, TAPTF-WC) have been mainly targeted for batch computing systems and therefore, not suitable for assigning tasks in time sharing

systems. Moreover, many existing policies (e.g. TAGS, TAGS-PM, TAPTF, TAPTF-WC, SITA-E, SITA-V, EQUILOAD) have not been designed to operate under changing traffic conditions and therefore, their performance degrade significantly under such conditions. We also note that many advanced task assignment policies are based on the assumption that the service time distribution of traffic closely follows heavy-tailed service time distributions. Although this is true for majority of cases, under specific cases when this cannot be justified, the performance of these advanced task assignment policies can be extremely poor.[7] Finally, existing task assignment policies have been designed to optimise the performance in individual server farms. As such, their performance is poor under multiple server farm environments as these policies do not exploit the properties of such environments. Based on these observations, we present and address four research questions in this thesis. Details of these research questions were presented in Chapter 1.

[7] We prove this in Chapter 6.

Chapter 3

Performance Modelling and Optimisation of a Time-sharing Server

This chapter investigates the way to optimise the performance in a time-sharing server by efficiently scheduling tasks.[1] We devote our attention to a particular time scheduling policy called the multi-level time sharing policy (MLTP). The rationale for concentrating on MLTP is because 1) MLTP requires no prior knowledge about actual task sizes and therefore, it can be used for scheduling a wide range of task types including dynamic web content and scientific workloads and 2) MLTP has shown significant performance improvements under distributions with the property of decreasing failure rate [Aalto et al., 2004; 2007], a key property of modern traffic represented by heavy-tailed distributions.[2] Due to these reasons, MLTP is used to schedule tasks in various systems such as routers and operating systems [Avrachenkov et al., 2004; Feng and Misra, 2003; Rai et al., 2005; 2004a; 2003; 2004b; Silberschatz et al., 1998].

Unfortunately, existing analytical models relating to MLTP have major limitations. For example, most of these studies present the analytical results under the assumption that the number of levels are infinite and the quanta are infinitely small [Aalto et al., 2007; 2005; 2004] because under these conditions it is relatively easy to model the behaviour of tasks under MLTP. However, neither infinite levels nor infinitely small quanta are practical or ever feasible to implement on real computer systems. A handful of studies that investigate the performance of MLTP under positive quanta are based on the assumption that both the inter-arrival times and service times follow exponential distributions [Coffman and Kleinrock, 1968]. Though the exponential inter-arrival times can be justified for a

[1] Note that the scheduling policies utilised in time-sharing computing systems are preemptive scheduling policies. We discussed preemptive scheduling policies at some length in Chapter 2.

[2] The decreasing failure rate simply means that the longer a task has run the longer it is expected to run.

range of scenarios [Cao et al., 2001; Williamson, 2001], extensive research carried out over the years, clearly indicates that the service times of computer workloads no longer follow exponential distributions but follow heavy-tailed distributions (for majority of cases) [Theophilus et al., 2010; Downey, 2005; 2001; Mitzenmacher, 2004; Cáceres et al., 1991; Arlitt and Williamson, 1996; Barford et al., 1999; Barford and Crovella, 1998; Leland and Ott, 1986; Arlitt and Williamson, 1997; Crovella and Bestavros, 1997; Harchol-Balter and Downey, 1997; Willinger et al., 1995]. In this chapter we investigate the performance of MLTP under heavy-tailed distributions under finite levels when the quanta are not infinitely small.

This chapter focuses on two types (variants) of MLTP, namely, multi-level optimal quantum time-sharing policy with N levels (N-MLTP-O) and multi-level equal quantum time-sharing policy with N levels (N-MLTP-E). The quanta for N-MLTP-O are computed to optimise a certain performance metric under a given set of conditions and the quanta for N-MLTP-E are equal in each level. As pointed out in Chapter 1, to evaluate the performance, we use two important performance metrics, namely, the expected waiting time and the expected slowdown.

The key contributions of this chapter are as follows.

- We derive the performance metrics (i.e. expected waiting time and expected slowdown) for MLTP and show that N-MLTP-O can result in significant performance improvements over N-MLTP-E and FCFS, particularly when both the system load and the task size variability are high.
- We investigate the impact of number of levels on the performance of both N-MLTP-O and N-MLTP-E. We show that as the number of levels increases, the performance of both policies increases, and the rate at which the performance increases depends on the factors such as the variability of service times and the system load.
- We investigate the behaviour of quanta for the case of two levels under different scenarios. We note that the (optimal) set of quanta that will result in optimal performance are unique for most of the scenarios. We also note that there is a sudden drop in quantum 1 that occurs between the system loads of 0.5 and 0.7 when the performance is evaluated using the expected time.
- We briefly discuss the impact of overestimating and underestimating the optimal quanta on the performance and discuss the measures that can be taken to minimise the degradation in the performance due to overestimating or underestimating optimal quanta.
- We investigate the performance of N-MLTP-E under a large number of queues. We use statistical regression modelling techniques to approximate the relationship between the performance

and the number of levels. For both performance metrics, we show that the relationship between the performance and the number of levels has a form of a power curve, where the two coefficients of the power curve are functions of both the variability of tasks and the system load.

- We compare the performance of N-MLTP-E with the performance of FB (refer to Section 2.2.2) and show that under highly variable traffic conditions, N-MLTP-E requires a large number of queues if it is to achieve the same performance levels as FB.

The rest of this chapter is organised as follows. Section 3.1 provides the details of M/G/1 MLTP. In Section 3.2 we discuss the way to compute the performance of MLTP under a Bounded Pareto service time distribution. Sections 3.3 and 3.4 present the analytical performance analysis of MLTP. The effect of quanta on performance is investigated in Section 3.5. In Sections 3.6 and 3.7 we investigate the fractions of tasks completed in levels and performance degradation in one performance metric as a result of optimising the performance using a different performance metric respectively. Section 3.8 investigates the performance of MLTP under a large number of queues. Section 3.9 compares the performance of MLTP with FB. The chapter is concluded in Section 3.10.

3.1 Multi-level Time Sharing Policy (MLTP)

MLTP has shown significant performance improvements over policies such as FCFS under realistic workload conditions. However, existing analytical analysis of MLTP has been carried out under very unrealistic conditions such as infinitely small quanta, infinite number of levels and exponential service time distributions. As discussed, this chapter investigates the performance of MLTP under heavy-tailed workloads (service time distributions) under finite number of levels when the quanta are not infinitely small. Such a policy is more consistent with those implemented on real computer systems and the findings will enable system designers to better understand how the factors such as the quanta, the system load, the number of levels and the task size variability will affect the performance of MLTP. Section 3.1.1 presents an overview of the MLTP model followed by the performance model for MLTP presented in Section 3.1.2.

3.1.1 Overview of Multi-level Time Sharing Policy (MLTP)

This section introduces the MLTP model. Figures 3.1 and 3.2 illustrate MLTP model. Note that these two representations are identical to each other. The notation provided in Table 3.1 is used to

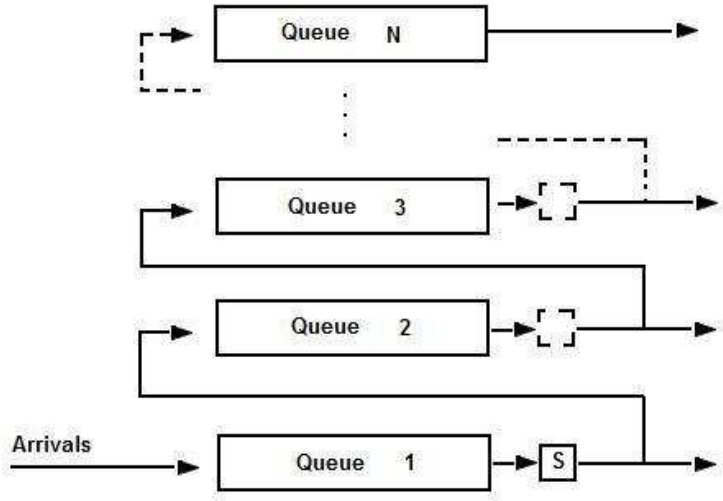


Figure 3.1: Multi-level Time Sharing Policy (MLTP): representation 1

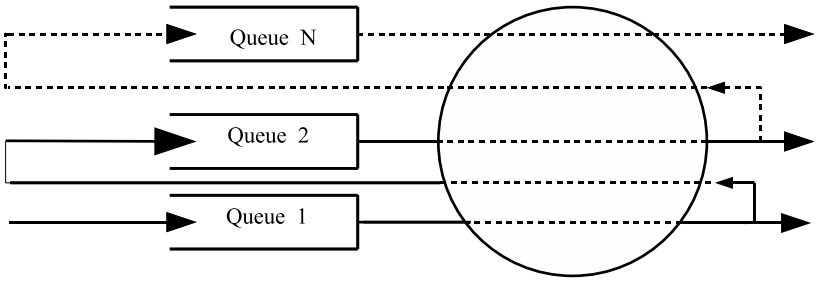


Figure 3.2: Multi-level Time Sharing Policy (MLTP): representation 2

describe MLTP. We note from Figures 3.1 and 3.2 that MLTP consists of N queues (levels). The basic functionality of MLTP is as follows.

- Each new task that arrives at the system is placed at the lowest level (queue), where the task is served in a FCFS manner until it receives maximum of q_1 amount of service (note: q_1 represents a time duration).
- If the service time of the task is less than or equal to q_1, the task departs system. Otherwise, the task is placed at Queue 2, where the task is processed in a FCFS manner until it receives at most q_2 amount of service and so on.

λ	Average arrival rate into Queue 1
T_i	i^{th} simple processing time (i.e. the length of processing time a task on Queue i)
$E[T_i]$	Expected value of T_i
$E[T_i^2]$	Second moment of T_i
$F(t)$	Cumulative distribution function (CDF) of service time distribution
U_i	Q_i if the task returns to the system at least $i-1$ times $T_1+T_2+...+T_k$ if the task returns the system only $k-1$ times where $k<i$
$E[U_i]$	Expected value of U_i
$E[U_i^2]$	Second moment of U_i
N	Number of queues (levels)
Λ_k	$\lambda(1-\int_0^{Q_i} dF(t))$
W_i	Waiting time in the system up to level i. This time does not include i simple processing times (i.e. $T_1, T_2, ..., T_i$)
q_i	Quantum (maximum processing time) i
Q_k	$q_1+q_2+.....+q_k$

Table 3.1: Notation: MLTP

- The task propagates through the system of queues until the total processing time, the task has so far received is equal to its service time at which point it leaves the system.
- A task waiting to be served in Queue i has the priority of service over tasks that are waiting to be served in Queue $i+1, i+2, ..., N$, where N denotes number of levels. However, a task currently being processed is not preempted upon the arrival of a new task to the system.

3.1.2 Performance model for MLTP

Here we present the details of the performance model for MLTP. The aim is to derive expressions for the expected waiting time and the expected slowdown. L. E. Schrage is one of the few researchers who attempted to model the behaviour of tasks in a MLTP system under an arbitrary service time distribution. In Schrage [1967], Schrage has derived an expression for the conditional expected flow-time of a task for MLTP under an arbitrary service distribution. More specifically, the expected flow-time of a task under MLTP given that the task's processing time is greater than Q_{i-1} and less than Q_i is given by

$$E[FT_i] = \frac{\lambda E[U_i^2] + \sum_{k=i+1}^{N} \Lambda_k E[T_k^2]}{2(1-\lambda E[U_{i-1}])(1-\lambda E[U_i])} + \frac{Q_{i-1}}{(1-\lambda E[U_{i-1}])} + E[T_i], \tag{3.1}$$

where $E[U_i]$, $E[U_{i-1}]$, $E[U_i^2]$, $E[T_k^2]$, $E[T_k]$, $E[T_k^2], E[T_k^{-1}]$ and Λ_k are given by (refer to [Schrage, 1967] for further information),

$$E[U_i] = \int_0^{Q_i} xdF(x) + Q_i(1-F(Q_i)), \tag{3.2}$$

$$E[U_{i-1}] = \int_0^{Q_{(i-1)}} xdF(x) + Q_{i-1}(1-F(Q_{i-1})), \tag{3.3}$$

$$E[U_i^2] = \int_0^{Q_i} x^2dF(x) + Q_i^2(1-F(Q_i)), \tag{3.4}$$

$$E[T_k] = \frac{1}{(1-F(Q_{k-1}))}(\int_{Q_k-1}^{Q_k}(x-Q_{k-1})dF(x) + q_k(1-F(Q_k))), \tag{3.5}$$

$$E[T_k^2] = \frac{1}{(1-F(Q_{k-1}))}(\int_{Q_k-1}^{Q_k}(x-Q_{k-1})^2dF(x) + q_k^2(1-F(Q_k))), \tag{3.6}$$

$$E[\frac{1}{T_k}] = \frac{1}{(1-F(Q_{k-1}))}(\int_{Q_k-1}^{Q_k}(x-Q_{k-1})^{-1}dF(x) + q_k^{-1}(1-F(Q_k))), \tag{3.7}$$

$$\Lambda_k = \lambda(1-\int_0^{Q_i} dF(t)). \tag{3.8}$$

We obtain the expected waiting time of a task under MLTP given that its processing time is greater than Q_{i-1} and less than Q_i by subtracting the expected total processing time up to Queue i from $E[FT_i]$. The expected total processing time up to $i-1$ queue is equal to Q_{i-1} and the expected processing time in Queue i is simply $E[T_i]$. Therefore, the expected waiting time of a task given that its service time is greater than Q_{i-1} and less than Q_i, $E[W_i]$, is given by

$$E[W_i] = \frac{\lambda E[U_i^2] + \sum_{k=i+1}^{N}\Lambda_k E[T_k^2]}{2(1-\lambda E[U_{i-1}])(1-\lambda E[U_i])} + \frac{Q_{i-1}}{(1-\lambda E[U_{i-1}])} - Q_{i-1}. \tag{3.9}$$

$E[W_i]$ defined above is strictly the waiting time of tasks in queues and it does not include the $i-1$ processing times (quanta).

Finally, we derive the expected slowdown of a task for MLTP given that its processing time is greater than Q_{i-1} and less than Q_i by multiplying $E[W_i]$ obtained above by $E[\frac{1}{Q_{i-1}+T_i}]$.

$$E[SD_i] = \left(\frac{\lambda E[U_i^2] + \sum_{k=i+1}^{N}\Lambda_k E[T_k^2]}{2(1-\lambda E[U_{i-1}])(1-\lambda E[U_i])} + \frac{Q_{i-1}}{(1-\lambda E[U_{i-1}])} - Q_{i-1}\right)E[(Q_{i-1}+T_i)^{-1}]. \tag{3.10}$$

Now that we have derived $E[W_i]$ (i.e. Equation 3.9) and $E[SD_i]$ (i.e. Equation 3.10), we can define the overall expected waiting time of a task for MLTP and the overall expected slowdown of a task for MLTP. For the sake of brevity, we simply refer to these as the expected waiting time and the expected slowdown. The expected time waiting can be obtained by multiplying $E[W_i]$ by the probability that a task's service time is greater than Q_{i-1} and less than Q_i and then taking the sum of all these terms as

follows:

$$E[W] = E[W_1]\int_0^{Q_1} f(x)dx + E[W_2]\int_{Q_1}^{Q_2} f(x)dx + ... + E[W_N]\int_{Q_{N-1}}^{Q_N} f(x)dx. \tag{3.11}$$

Similarly, we obtain the expected slowdown as follows:

$$E[SD] = E[SD_1]\int_0^{Q_1} f(x)dx + E[SD_2]\int_{Q_1}^{Q_2} f(x)dx + ... + E[SD_N]\int_{Q_{N-1}}^{Q_N} f(x)dx. \tag{3.12}$$

The expected waiting time/slowdown for N-MLTP-O is computed by minimising the expected waiting time/slowdown given by above equations under a given α (i.e. task size variability), system load and N.[3] The expected time for N-MLTP-E is computed by substituting $q_1 = q_2 = ... = q_N = \frac{10^7}{N}$ under a given α value, system load and N into above equation. The expected waiting time under FCFS is obtained using the Pollaczek-Khinchin formula [Kleinrock, 1975].

3.2 The use of MLTP to schedule tasks with Bounded Pareto service time distributions

Bounded Pareto distribution is one of the most commonly used [Harchol-Balter, 2002; Broberg et al., 2004; 2006; Harchol-Balter et al., 1999; Tari et al., 2005; Fu and Tari, 2003; Crovella et al., 1998a] heavy-tailed distributions, which is used for developing analytically tractable performance models. Here we discuss the way to model the performance of MLTP under a Bounded Pareto distribution.

In the case of Bounded Pareto distributions, we let

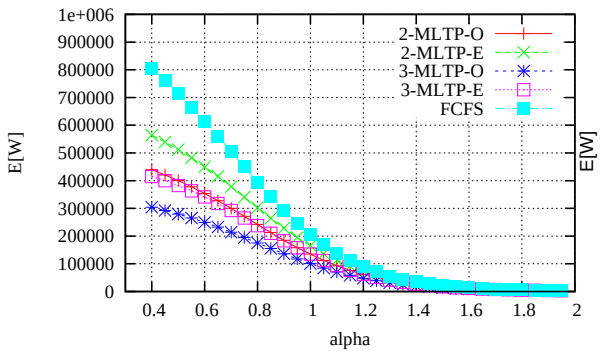
$$Q_N = q_1 + q_2 + .. + q_N = p, \tag{3.13}$$

where p represents the upper bound of the service time distribution. Q_N and N represent the sum of quanta up to level N and the (total) number of levels respectively. Both performance metrics (i.e. $E[W]$ and $E[SD]$) are dependent on q_i $(0 < i \leq N)$, k, p, α, λ and N. In the next section we will evaluate the expected waiting time and expected slowdown for multi-level time sharing policy under a range of workloads and task size variabilities.

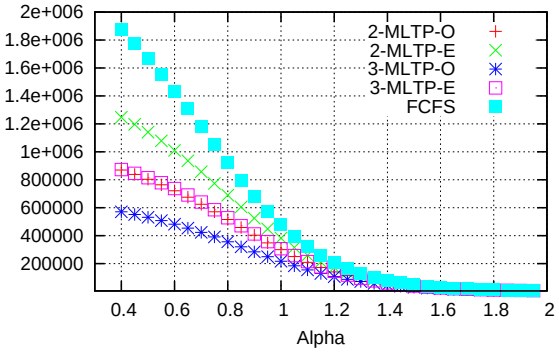
3.3 Analysis of expected waiting time

The expected waiting time is the most widely used performance metric for evaluating the performance of scheduling policies. This section investigates the expected waiting time for 2-MLTP-O, 2-MLTP-E, 3-MLTP-O and 3-MLTP-E under a range of scenarios. Figure 3.3 plots the expected waiting time for policies and Figure 3.4 plots the factor of improvement in MLTP over FCFS.

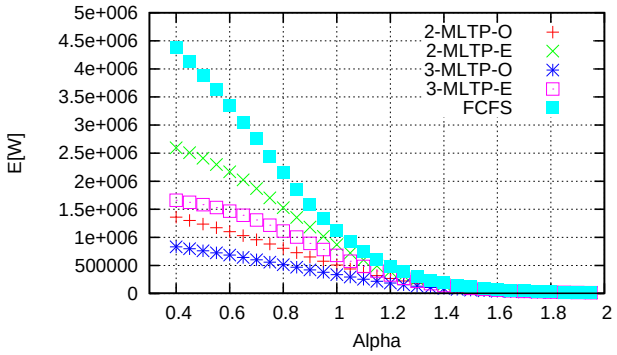
[3] These optimisation problems are solved using Mathamatica [Wolfram Research, 2003].



(a) System load = 0.3



(b) System load = 0.5

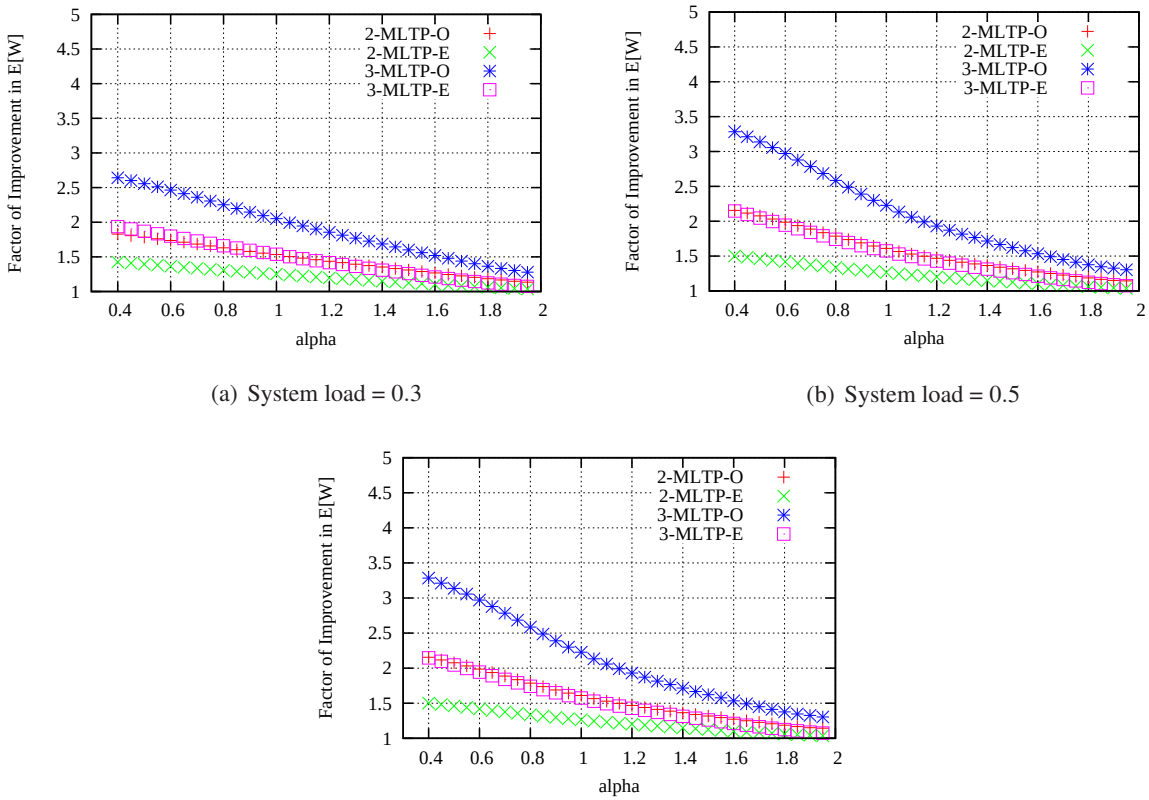


(c) System load = 0.7

Figure 3.3: Expected waiting time for MLTP with two and three queues

Let us first discuss the performance of 2-MLTP-O, 2-MLTP-E and FCFS. We note from Figure 3.3 that 2-MLTP-O outperforms both 2-MLTP-E and FCFS under all the scenarios considered. For example, under a system load of 0.7, when $\alpha = 0.4$, 2-MLTP-O outperforms FCFS and 2-MLTP-E by factors of 3 and 2 respectively. Under the same system load, when α is equal to 1.1, 2-MLTP-O outperforms FCFS and 2-MLTP-E by factors of 2 and 1.5 respectively. We note that the factor of improvement is highly significant when both the system load and the task size variability are high (i.e. low α). On the other hand, if both the system load and task size variability are low (i.e. high α), then the factor of improvement is not highly significant.

Let us now consider the expected waiting time for 3-MLTP-O, 3-MLTP-E and FCFS. We note that 3-MLTP-O outperforms both 3-MLTP-E and FCFS. As was noted for 2-MLTP-O and 2-MLTP-E, the factor of improvement is inversely proportional to α and directly proportional to the system load. The factor of improvement is highly significant under high system loads and high task size variabilities.



(a) System load = 0.3

(b) System load = 0.5

(c) System load = 0.7

Figure 3.4: Factor of improvement in expected waiting time for MLTP with two and three queues

For example, under a system load of 0.7, when α is equal to 0.4, 3-MLTP-O performs 5 times better than FCFS and 2 times better than 3-MLTP-E. We note that FCFS has the worst performance for all the cases since FCFS does nothing to reduce the variance of task sizes in host queues, nor does it gives preferential treatment to small tasks.

Let us now look at the effect of number of levels on the expected waiting time. We note from Figures 3.3 and 3.4 that 3-MLTP-O outperforms 2-MLTP-O under all scenarios (i.e. system loads and task size variabilities). Furthermore, we note that 3-MLTP-E outperforms 2-MLTP-E under all scenarios. As α increases, the factor of improvement in 3-MLTP-O over 2-MLTP-O decreases. For example, under a system load of 0.7, when α is equal to 0.4, 3-MLTP-O performs 1.6 times better than 2-MLTP-O. Under the same system load, when $\alpha = 1.1$, 3-MLTP-O outperforms 2-MLTP-O by a factor of 1.4. We note that if the variability of task sizes is high, we can improve the performance of MLTP by increasing the number of levels. However, when the variability of tasks is very low, an

increase in the number of levels does not result in significant performance improvements.

Let us now discuss why there is an improvement in the expected waiting time with the number of levels. Recall that under MLTP, when a task on a particular level is being serviced, a new task that arrives at the system (into Queue 1) cannot interrupt the (currently processing) task. Therefore, as far as individual levels are concerned, tasks are serviced in a FCFS manner until the quantum expires. In this type of a system expected waiting time in a particular queue is proportional to the variance of task sizes waiting in the queue. As the number of levels increases, variance of task sizes in queues decreases resulting in an improvement in the overall expected waiting time.

Finally, we note that 3-MLTP-E performs slightly better than 2-MLTP-O under low system loads. However, as the system load increases 2-MLTP-O begins to outperform 3-MLTP-E. Under a system load of 0.7, when α is low, we note that 2-MLTP-O performs significantly better than 3-MLTP-E. We also note that when the system load is high and α is low, the expected waiting time for 5-MLTP-E is equivalent to the expected waiting time for 2-MLTP-O.

3.3.1 Load in queues

By analysing the load in queues under N-MLTP-O and N-MLTP-E we can justify the performance improvements in N-MLTP-O over N-MLTP-E. Here we investigate the load in queues (levels) under a range of scenarios. We compute the load in level i by multiplying the average arrival rate into level i (given by Equation 3.8) by $E[T_i]$, where $E[T_i]$ is the expected value of i^{th} processing time (given by Equation 3.5). Let $load_i$ be the load in level i. Then,

$$load_i = \Lambda_i * E[T_i]. \tag{3.14}$$

Note that the sum of loads in levels is equal to the system load,

$$load = load_1 + load_2 + ... + load_N. \tag{3.15}$$

Let $load_i\%$ be the load in level i as a percentage of the system load, i.e. ($load_i$/system load)*100%. Figure 3.5 illustrates the behaviour of $load_i\%$ with α for 2-MLTP-O and 2-MLTP-E.

The main observations for the case of two levels are as follows.

- Under low and moderate system loads, the shapes of load curve are similar for both N-MLTP-O and N-MLTP-E. However, the difference between $load_1\%$ and $load_2\%$ is higher for 2-MLTP-E compared to that of 2-MLTP-O for all α values considered.

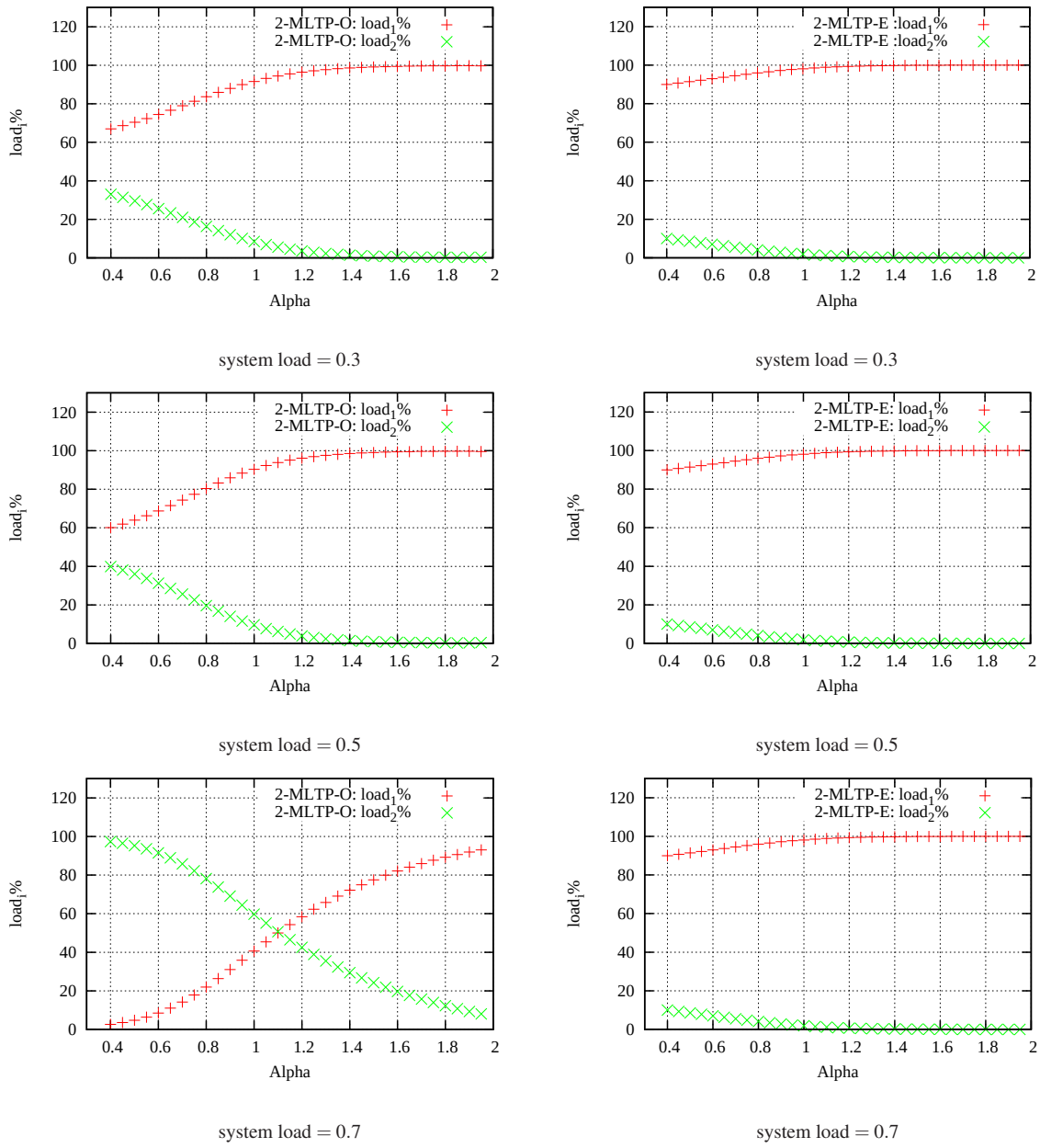


Figure 3.5: Load in queues for 2-MLTP-O and 2-MLTP-E with optimal expected waiting time

- There is a significant change in the shape of load queues for 2-MLTP-O as the system load changes from moderate to high, which is not the case for 2-MLTP-E.
- 2-MLTP-O improves the performance by unbalancing the load among the two levels. For

example, under a system load of 0.3, when α is equal to 0.4, the ratio between $load_1\%$ and $load_2\%$ is equal to 2, while under the same system load, when α equal to 1.1, the ratio between $load_1\%$ and $load_2\%$ is 16.

- Although 2-MLTP-E does unbalance the load among its levels, 2-MLTP-E does not perform well because it does not unbalance the load in an optimal manner. For example, under a system load of 0.3, when α is equal to 0.4, the ratio between $load_1\%$ and $load_2\%$ equals 9, while under the same system load, when α is equal to 1.1, the ratio between $load_1\%$ and $load_2\%$ is 84.

Let us now briefly investigate the behaviour of loads under 3-MLTP-O and 3-MLTP-E. Figure 3.6, illustrates $load_i\%$ under 3-MLTP-O and 3-MLTP-E under three different system loads. Clearly, there are similarities between the behaviour of load between two and three levels. The main observations for the case of three levels are listed below.

- $load_1\%$ continuously increases with α for both 3-MLTP-O and 3-MLTP-E.
- $load_1\%$ and $load_2\%$ continuously decrease with α for both 3-MLTP-O and 3-MLTP-E.
- For 3-MLTP-E $load_1\% > load_2\% > load_3\%$ under all system loads.
- For 3-MLTP-O $load_1\% > load_2\% > load_3\%$ under low and moderate system loads.

3.4 Analysis of expected slowdown

The slowdown measures the fairness of a scheduling policy under a given scheduling policy. Here we investigate the expected slowdown for MLTP under a range of workload scenarios. Most of our observations in this section are similar to what we already saw in the previous section for the expected waiting time. However, there are a few important differences. The main focus of this section is to point out these differences. Figure 3.7 plots the expected slowdown for MLTP and the factor of improvement in expected slowdown over FCFS.

We note that the factor of improvement in the expected slowdown under 2-MLTP-O is higher than the factor of improvement in expected waiting time under 2-MLTP-O. For example, under a system load of 0.7, when $\alpha = 0.4$, the factor of improvement in the expected slowdown is equal to 4.5. Under the same conditions, the factor of improvement in expected waiting time is equal to 3. We also note from Figure 3.7 that 2-MLTP-O outperforms 3-MLTP-E for all the scenarios considered.

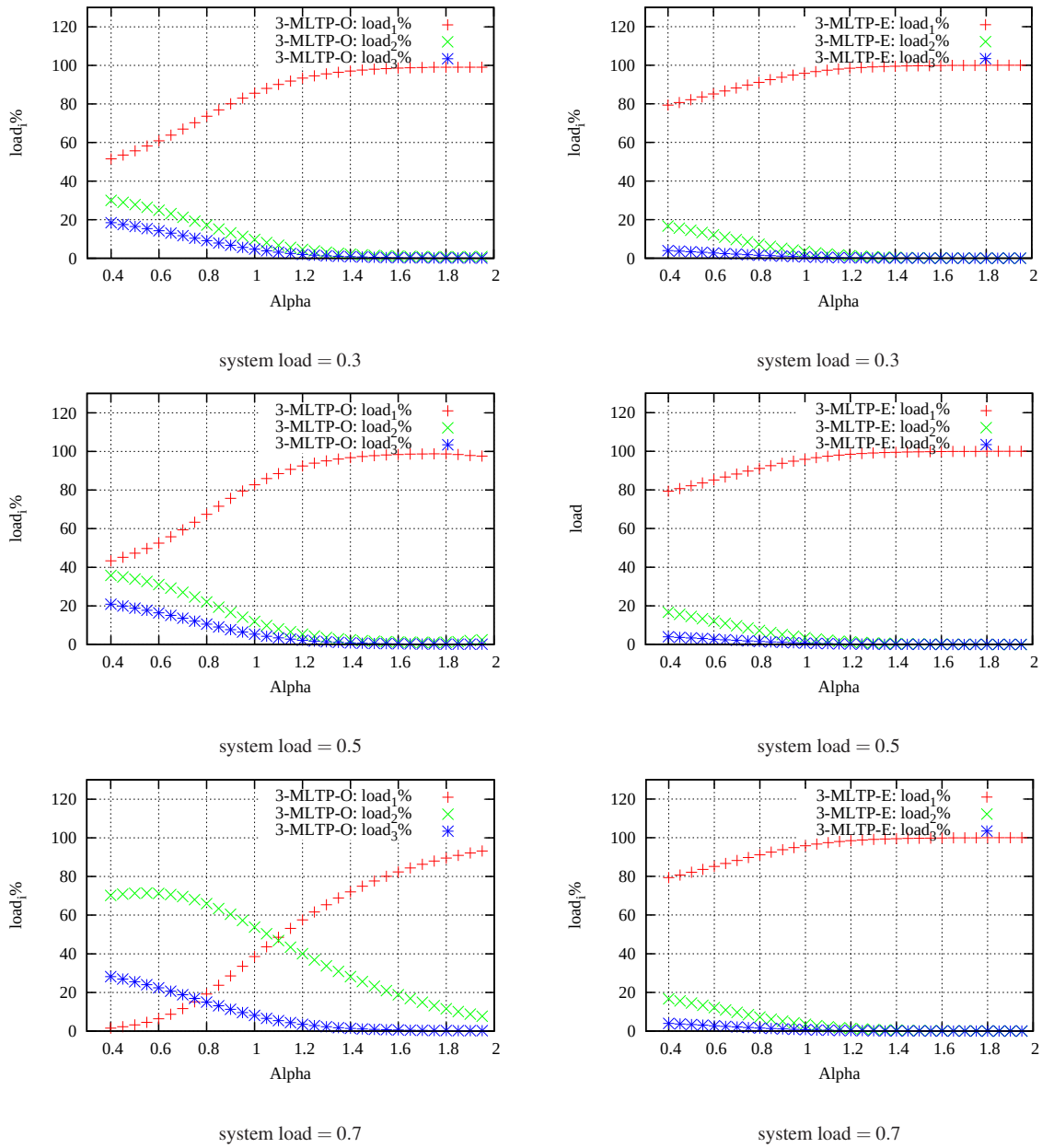


Figure 3.6: Load in queues for 3-MLTP-O and 3-MLTP-E with optimal expected waiting time

This is different from what we noted in the previous section for expected waiting time, where we saw 3-MLTP-E outperforming 2-MLTP-O under certain cases.

Let us now briefly investigate the load in queues when the quanta are computed to optimise the expected slowdown. Figure 3.8 shows the behaviour of load. As far as 2-MLTP-O is concerned,

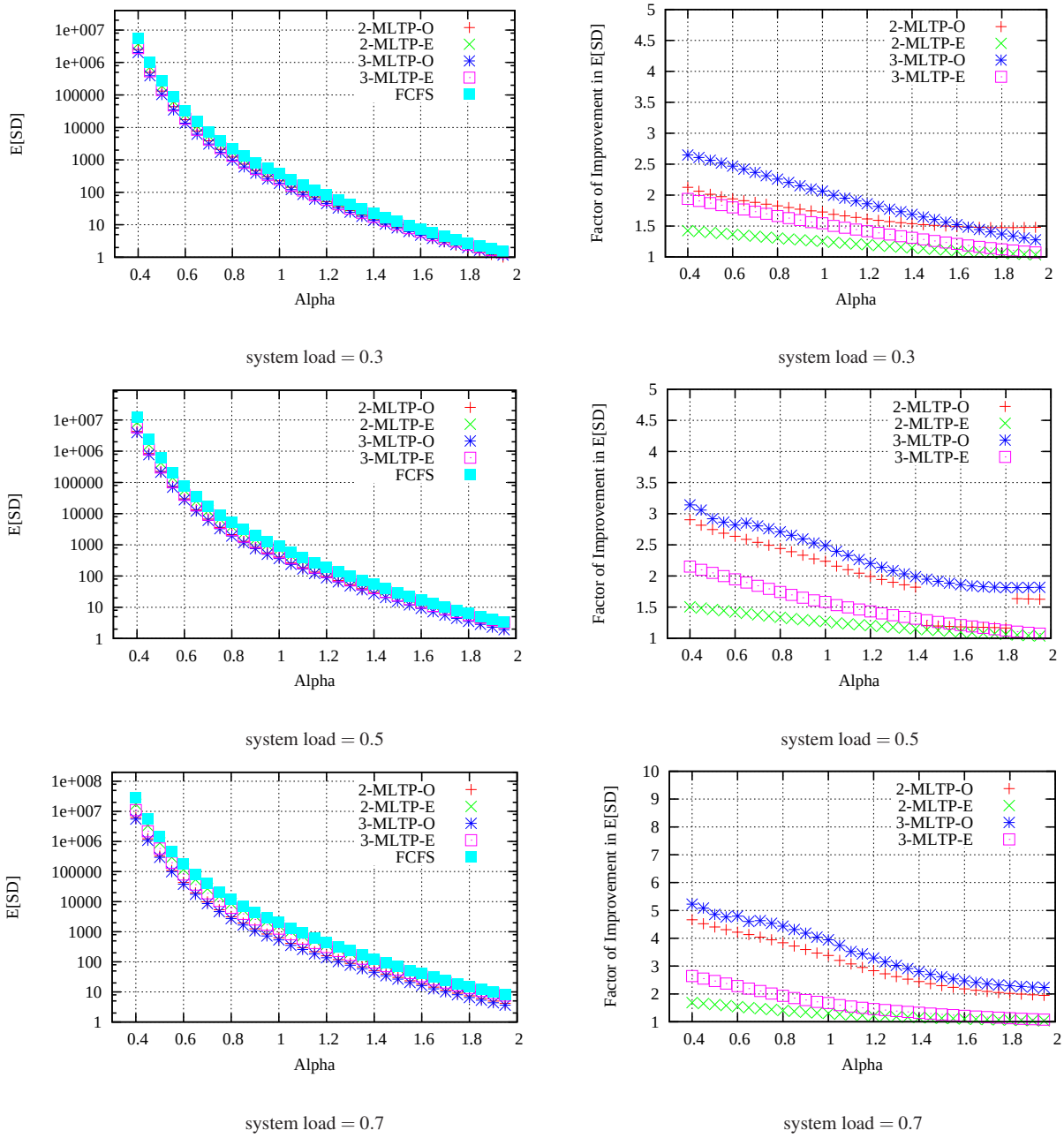


Figure 3.7: Expected slowdown and factor of improvement in expected slowdown for MLTP with two and three queues

we see that $load_1\%$ is very low, when α is low and as α increases, $load_1\%$ increases, but $load_2\%$ decreases. In the case of 3-MLTP-O both $load_1\%$ and $load_2\%$ are very small, but increase with

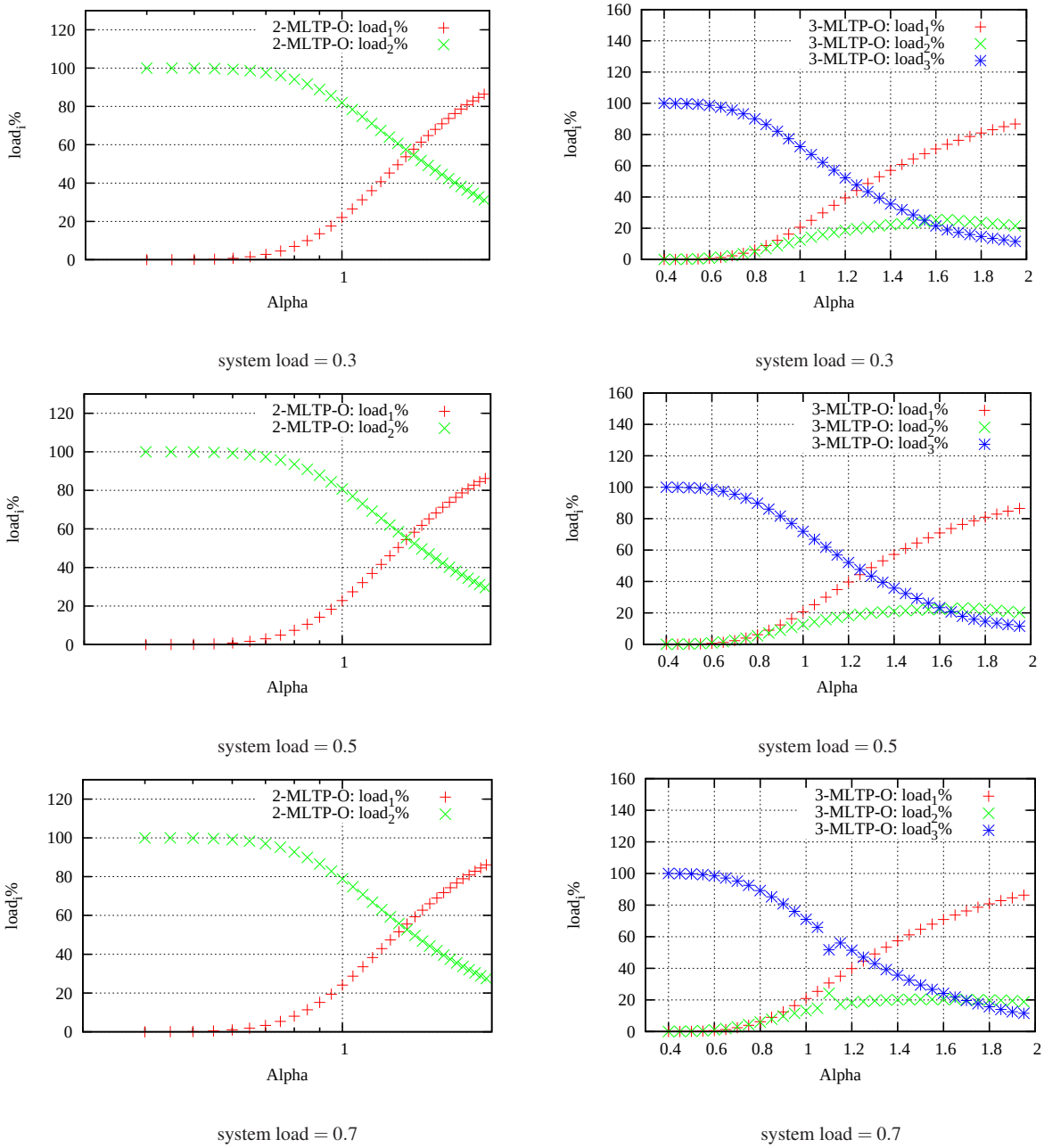


Figure 3.8: Load in queues for 2-MLTP-O and 3-MLTP-O with optimal expected slowdown

increasing α. In the previous section we saw (for the case of expected waiting time) that when the system load changes from moderate to high, the behaviour of loads in levels changes significantly. For the expected slowdown such changes are not observed. We can clearly see that N-MLTP-O improves

expected slowdown by unbalancing the load. In previous studies [Crovella et al., 1998a] it has been shown that the unbalancing the loads among the servers often improves the expected slowdown.

3.5 Behaviour of cut-offs under MLTP-O

This section looks at the behaviour of quanta for N-MLTP-O under different workload scenarios. The aim is to get an idea of which quanta (sizes) are more suitable for which type of workloads.

Note that the quanta for 2-MLTP-O are computed to optimise either the expected waiting time or the expected slowdown. In this section we show that when the number of levels is equal to 2, the optimal set of quanta are unique for most of the scenarios (i.e. task size variabilities and system loads) for both performance metrics. For some system loads, there exist two sets of quanta that may result in near optimal performance. In such cases a system designer may use either set of quanta. However, if the system designer is not certain about the exact values of quanta to be used, we recommend that he/she uses the set of quanta that is least sensitive to the performance. We will discuss this in detail later in this section.

To simplify the problem, we transform the quantum based multi-level time sharing system into a cut-off based multi-level time sharing system by partitioning the domain $[0, p]$ of the Bounded Pareto distribution into a series of cut-off points $p_1, p_2, ..., p_N$. The relationship between quanta and cut-offs are such that

$$\begin{aligned} p_1 &= q_1, \\ p_i &= q_1 + q_2 + .. + q_i, \\ P_N &= p, \end{aligned} \tag{3.16}$$

where N and p denote the number of levels and the upper bound of the service time distribution respectively.

3.5.1 Effect of cut-offs on the performance of MLTP: $N = 2$

Here we investigate the effect of cut-offs on the performance when the number of levels is equal to 2. Figures 3.9, 3.10 and 3.11 illustrate the effect of p_1 on the expected waiting for the case of two levels.

Let $p_{i\_w\_opt}$ be the cut-off i that results in the minimum expected waiting time. The key findings are as follows.

- Under a fixed system load, an increase in α results in $p_{i\_w\_opt}$ to decrease. For example, under

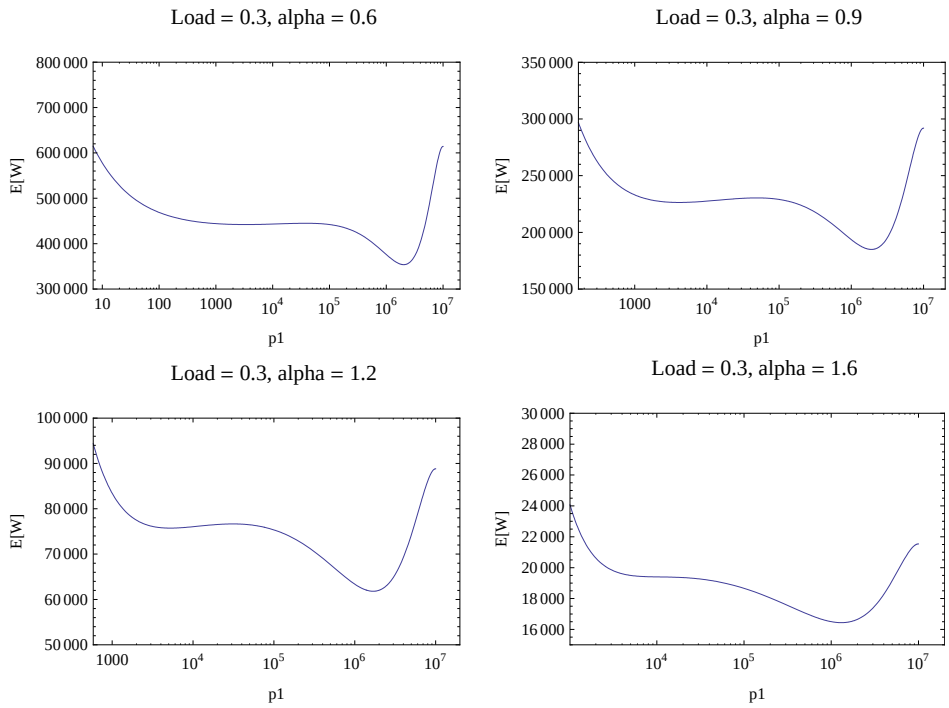


Figure 3.9: Impact of p1 on the expected waiting time of MLTP: system load = 0.3

a system load of 0.3, as α increases from 0.4 to 1.8, $p_{i\_w\_opt}$ decreases approximately from 2.0 x 10^6 to 3 x 10^5.

- Under low and moderate system loads, $p_{i\_w\_opt}$ is significantly higher than that of high system. loads.

- The curves have 2 minima for most of the system loads and α values considered. Moreover, for some system loads and task size variabilities (e.g. system load $= 0.5$ and $\alpha = 0.5$), these two minima are very close to each other. In this case one can use either p_1 as each p_1 results in similar performance. However, if the exact value of p_1 is not known it is better to use p_1 with the higher value as it is relatively less sensitive to the expected waiting time. This ensures that there are no significant performance degradations by slightly overestimating or underestimating the optimal p_1.

- Figure 3.12 illustrates the behaviour of $p_{1\_w\_opt}$ with the system load. We note that there is a sudden drop in $p_{1\_w\_opt}$, which occurs between the system loads of 0.6 and 0.7. This drop in $p_{1\_w\_opt}$ justifies our previous observations in Figures 3.9, 3.10 and 3.11.

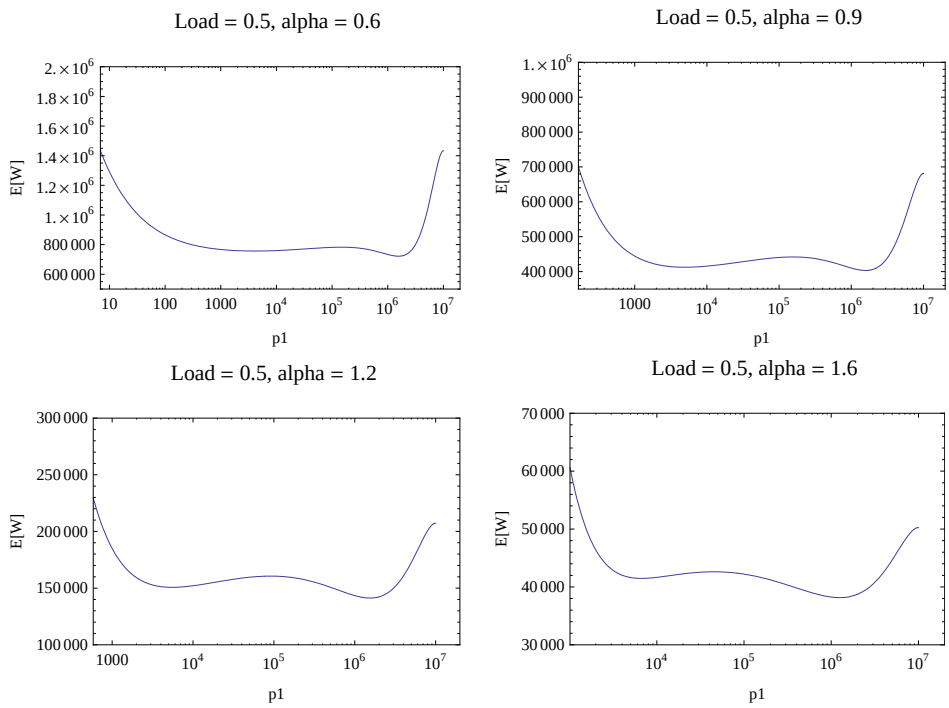


Figure 3.10: Impact of p1 on the expected waiting of MLTP: system load = 0.5

Let us now briefly look at the impact of the p_1 on the expected slowdown of MLTP. Figures 3.13, 3.14 and 3.15 plot p_1 vs expected slowdown.

Let $p_{i\_sd\_opt}$ be cut-off i corresponding to the minimum expected slowdown. As we noted before, for the case of expected waiting time, we see that these plots consist of two minima. However, contrary to our previous observations in relation to the expected waiting time, the optimal p_1 corresponding to the least expected slowdown (i.e. $p_{i\_sd\_opt}$) always correspond to the minima on the left. We also note that $p_{i\_sd\_opt}$ is very small compared to the largest task, p of the service time distribution for all the scenarios considered.

3.5.2 Effect of cut-offs on the performance of MLTP: $N > 2$

This section briefly discusses the behaviour of cut-offs when the number of levels is equal to 3. Figures 3.16 and 3.17 illustrate the effect of p_1 and p_2 on the expected waiting time and the expected slowdown under two specific scenarios.

We note that in each plot there is a (global) minimum, which corresponds to the optimal expected waiting time and the optimal expected slowdown. In fact, there are multiple sets of p_1 and p_2 that

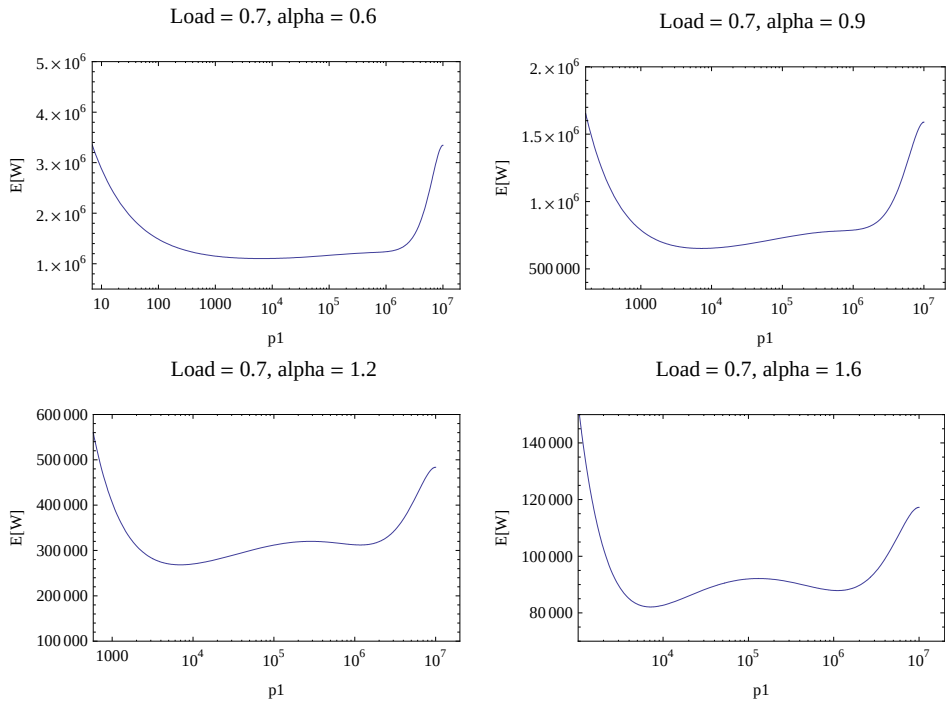


Figure 3.11: Impact of p1 on the expected waiting time of MLTP: system load = 0.7

result in near optimal performance. In such cases, Mathematica [Wolfram Research, 2003] does not always return the global minimum. However, it is possible to obtain the global minimum by specifying local boundaries of the global solution as a constraint in the Mathematica optimisation function. These boundaries can be obtained using plots such as 3.16 and 3.17.

3.6 Fraction of tasks completed on levels in 2-MLTP-O

This section looks at the fraction of tasks completed in levels under 2-MLTP-O. The aim is to identify the unique behaviours in the fraction of task completed in levels under different workload conditions. Such information is useful when making scheduling decisions for N-MLTP-O.

The fractions of tasks completed in levels are computed using the cumulative distribution function of the Bounded Pareto distribution denoted by $F(x)$. Let us investigate the fractions of tasks completed in levels under 2-MLTP-O when p_1 is computed to optimise the expected waiting time. Let $frac\_l1\_ew$ and $frac\_l2\_ew$ be the fractions of tasks completed in level 1 and level 2 respectively.

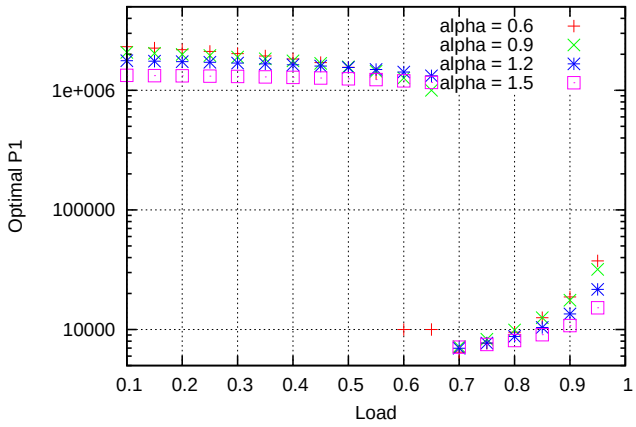


Figure 3.12: Behaviour of optimal p_1 for 2-MLTP-O

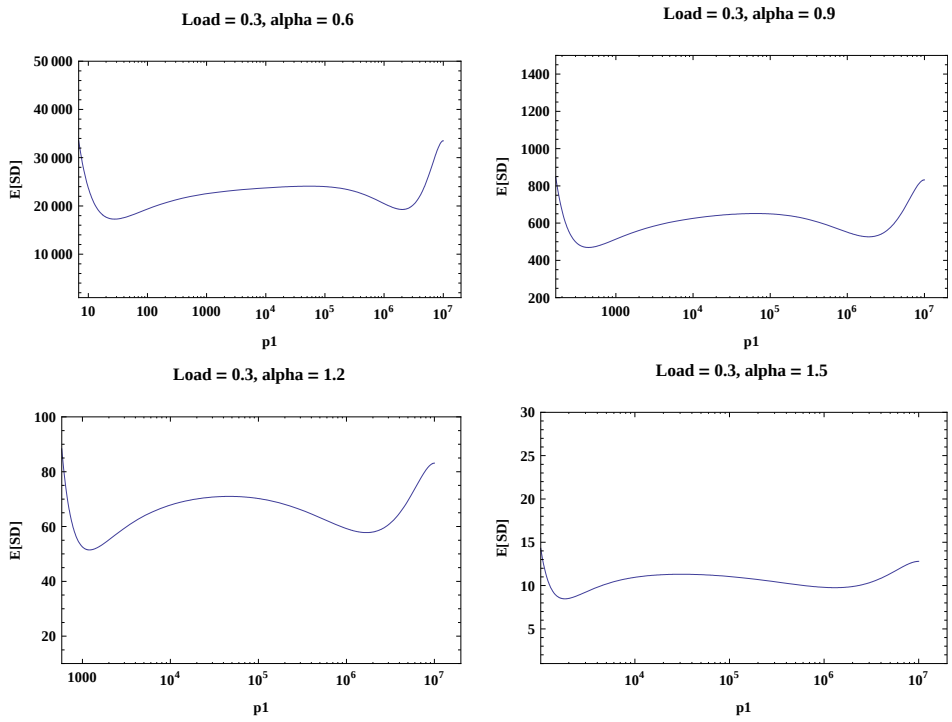


Figure 3.13: Impact of p1 on the expected slowdown of MLTP: system load = 0.3

Then,

$$frac\_l1\_ew = F(p_{1\_w\_opt}),$$
$$frac\_l2\_ew = 1 - F(p_{1\_w\_opt}), \tag{3.17}$$

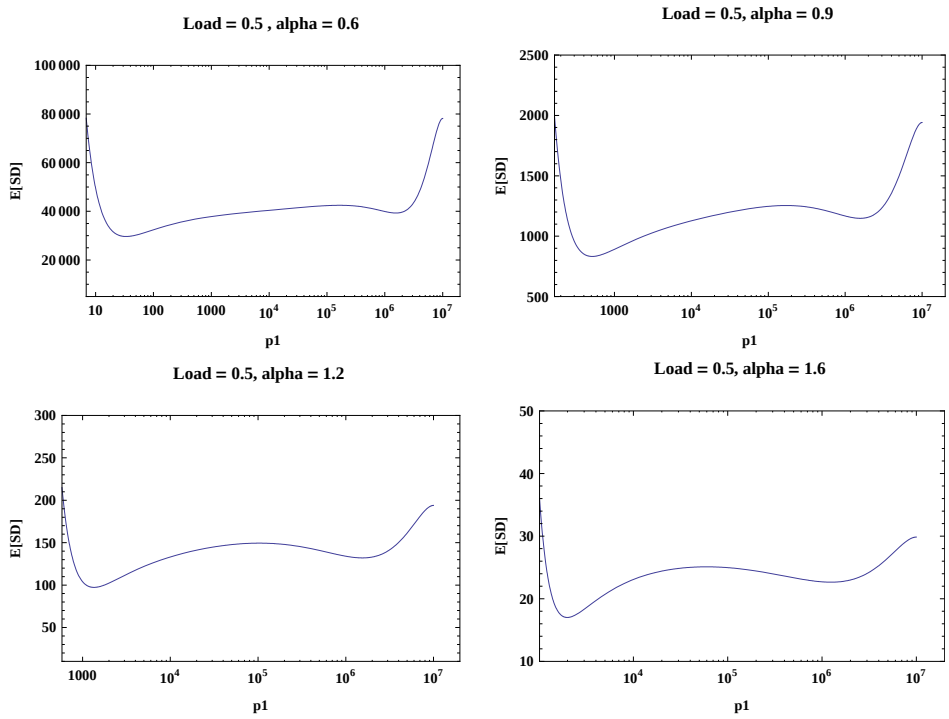


Figure 3.14: Impact of p1 on the expected slowdown of MLTP: system load = 0.5

where $p_{1\_w\_opt}$ denotes the value of p_1 when p_1 is computed to optimise the expected waiting time. Figure 3.18 shows the fractions of tasks completed in level 1 under four different system loads. We note that more than 95% of tasks are completed in level 1 for all the scenarios considered. Under low and moderate system loads, the fraction of tasks completed in level 1 is as high as 99.99%. As the system load increases, the fraction of tasks completed in level 1 decreases by a very small amount (5%).

Let us now consider the fractions of tasks completed in levels under MLTP when p_1 is computed to optimise the expected slowdown. Let $frac\_l1\_sd$ and $frac\_l2\_sd$ be the fractions of tasks completed in level 1 and level 2 respectively. Then,

$$\begin{aligned} frac\_l1\_sd &= F(p_{1\_sd\_opt}), \\ frac\_l2\_sd &= 1 - F(p_{1\_sd\_opt}), \end{aligned} \tag{3.18}$$

where $p_{1\_sd\_opt}$ denotes the value of p_1 when p_1 is computed to optimise the expected slowdown. Figure 3.19 illustrates the fractions of tasks completed in level 1 under four different system loads. We note that $frac\_l1\_sd$ is not as high as $frac\_l1\_ew$. The highest value of $frac\_l1\_sd$ is about

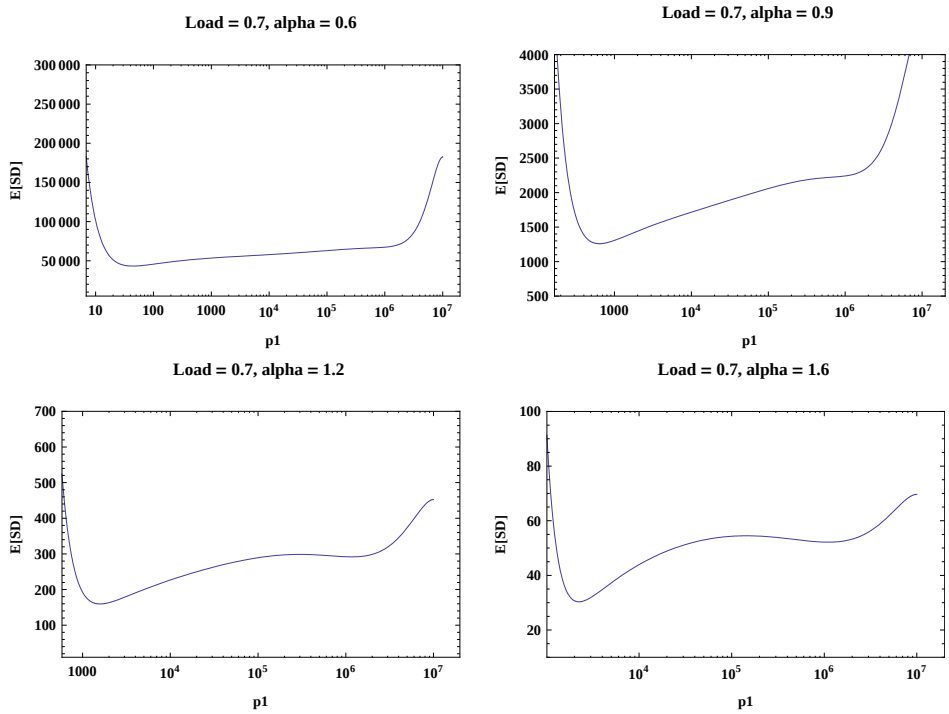


Figure 3.15: Impact of p1 on the expected slowdown of MLTP: system load = 0.7

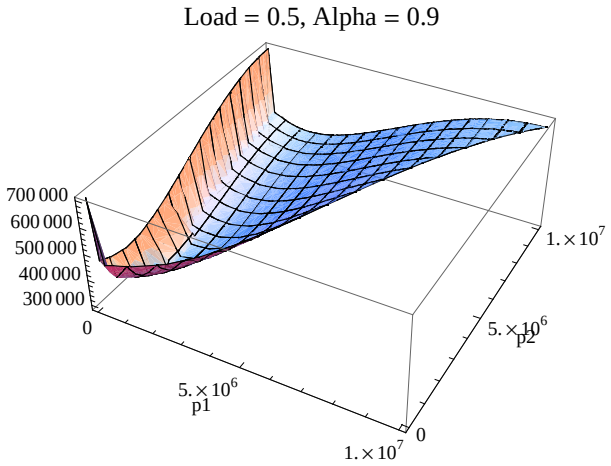


Figure 3.16: Effect of p1 and p2 on the expected waiting time of MLTP: system load = 0.5 and α = 0.9

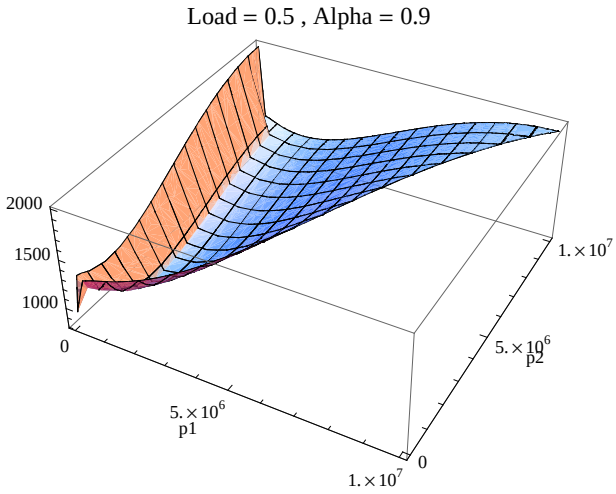


Figure 3.17: Effect of p1 and p2 on the expected slowdown of MLTP: system load = 0.5 and α = 0.9

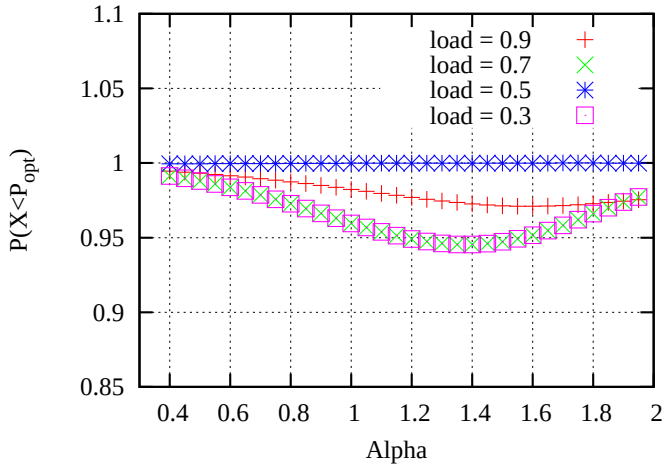


Figure 3.18: Behaviour of $frac\_l1\_ew$ for 2-MLTP-O

80%. In the case of low and moderate system loads, the value of $frac\_l1\_sd$ is less than 70%. As the system load increases, $frac\_l1\_sd$ tends to increase. We also note that under a constant system load, the fractions of task completed in levels do not vary at large, if α lies in the range 0.8 and 1.6. This means in this α range, once $p_{1\_sd\_opt}$ is computed for a given α value, $p_{1\_opt\_sd}$ for other α values can be computed simply by substituting $p_{1\_sd\_opt}$, α, p and k values into the inverse cumulative distribution function of the Bounded Pareto distribution. When designing systems that utilise adaptive

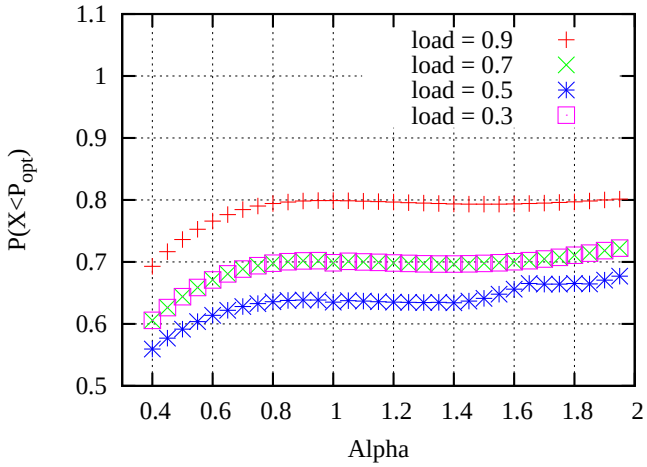


Figure 3.19: Behaviour of frac_l1_sd for 2-MLTP-O

(optimal cut-offs are computed online) multi-level time sharing policies, such a method can be very useful as it allows optimal cut-offs to be computed without having to solve complex optimisation problems multiple times.

3.7 Degradation in performance

The performance evaluations of scheduling and task assignment policies typically involve deriving a performance metric (as a function of various scheduling parameters) by applying queueing theoretic fundamentals and then computing the best parameters by optimising this performance metric. In general, it is not possible to optimise two different performance metrics at the same time and it is often the case that optimising one performance metric can lead to degradation in another performance metric. Unfortunately, existing work does not investigate this problem in detail. In this section we investigate the degradation in the expected waiting time when cut-offs for 2-MLTP-O are computed to optimise the expected slowdown and vice versa. Let $E[W]_{Deg}\%$ be the degradation in the expected waiting time when the cut-offs for 2-MLTP-O are computed to optimise the expected slowdown. Then,

$$E[W]_{Deg}\% = \frac{E[W]_{p_{1\_sd\_opt}} - E[W]_{p_{1\_w\_opt}}}{E[W]_{p_{1\_w\_opt}}}, \tag{3.19}$$

where $E[W]_{p_{i\_sd\_opt}}$ denotes the expected waiting time for 2-MLTP-O when the cut-offs are computed to optimise the expected slowdown. $E[W]_{p_{i\_w\_opt}}$ denotes the optimal expected waiting time for 2-MLTP-O.

Similarly, we define the percentage performance degradation in the expected slowdown, $E[SD]_{Deg}\%$ as follows:

$$E[SD]_{Deg}\% = \frac{E[SD]_{p_{1\_w\_opt}} - E[SD]_{p_{1\_sd\_opt}}}{E[SD]_{p_{1\_sd\_opt}}}, \tag{3.20}$$

where $E[SD]_{p_{1\_w\_opt}}$ represents the expected slowdown under 2-MLTP-O when the cut-offs are computed to optimise the expected waiting time. $E[SD]_{p_{1\_sd\_opt}}$ denotes the optimal expected slowdown for 2-MLTP-O.

Table 3.2 illustrates the degradation in $E[W]$ and $E[SD]$ under four different system loads. We

Table 3.2: Degradation in E[SD] and E[W] for MLTP with two queues

α	system load	$E[SD]_{Deg}\%$	$E[W]_{Deg}\%$
0.4	0.3	20%	50%
0.8	0.3	15%	35%
1.2	0.3	18%	30%
1.6	0.3	20%	25%
2.0	0.3	30%	30%
0.4	0.5	38%	50%
0.8	0.5	40%	25%
1.2	0.5	38%	20%
1.6	0.5	40%	10%
2.0	0.5	42%	10%
0.4	0.7	35%	80%
0.8	0.7	30%	50%
1.2	0.7	38%	30%
1.6	0.7	30%	25%
2.0	0.7	40%	20%
0.4	0.9	41%	250%
0.8	0.9	40%	100%
1.2	0.9	50%	60%
1.6	0.9	60%	45%
2.0	0.9	61%	40%

note that under high system loads and high task size variabilities (i.e. low α values), $E[W]_{Deg}\%$ is very high. For example, under a system load of 0.9, when α equals 0.4, $E[W]_{Deg}\%$ is equal to 250%. $E[W]_{Deg}\%$ decreases consistently with increasing α.

We note that $E[SD]_{Deg}\%$ lies in the range of 10%- 60% for all system loads and task size variabil-

ities considered. In general, relatively small p_1 improves both the expected slowdown and expected waiting time. However, the use of very small p_1 to optimise the expected slowdown can result in expected waiting time to deteriorate significantly (250%).

3.8 Performance of N-MLTP-E under a large N

Previous sections investigated various properties of MLTP when the number of levels are equal to 2 and 3. We noted that that there is an improvement in both the expected waiting time and expected slowdown with the (increasing) number of levels. Here we investigate the expected waiting time and expected slowdown under N-MLTP-E up to one hundred levels. The aim is to study the effect of number of levels on the performance. Here we show that under a given system load and a task size variability, the relationship between the performance and the number levels can be accurately modelled using a power curve. We investigate the behaviour of the two constants of the power curve under different N for each performance metric and show that these two coefficients are functions of both α and the system load.

Here we do not consider N-MLTP-O, rather we focus only on N-MLTP-E due to two reasons. First as the number of levels increases, computing the performance of N-MLTP-O becomes very difficult, because in order to compute the optimal performance we need to solve highly complex optimisation problems, which cannot be solved using Mathematica. Second as N increases, the factor of improvement in N-MLTP-O over N-MLTP-E decreases. This means that if N is large then the performance of two N-MLTP-O and N-MLTP-E is not significantly different. The cut-offs for N-MLTP-E are computed as

$$p_i = i * (10^7/N), \tag{3.21}$$

where p_i denotes the i^{th} cut-off and N denotes the number of levels.

3.8.1 Expected waiting time for MLTP-E under a large N

Let us investigate the expected waiting time for N-MLTP-E, where $(0 < N \leq 100]$. We compute the expected waiting time for N-MLTP-E under 32 different α $(0.4 \leq \alpha \leq 1.95)$ values and four different system loads. In doing so, we get 128 different data sets each of which we fit a power curve regression model and then investigate coefficient of determination. We note that the coefficient of determination (r^2) of (all of) these power curves are very close to 1 (greater than 0.99). This means that the relationship between the expected waiting time and the number of levels can be described in

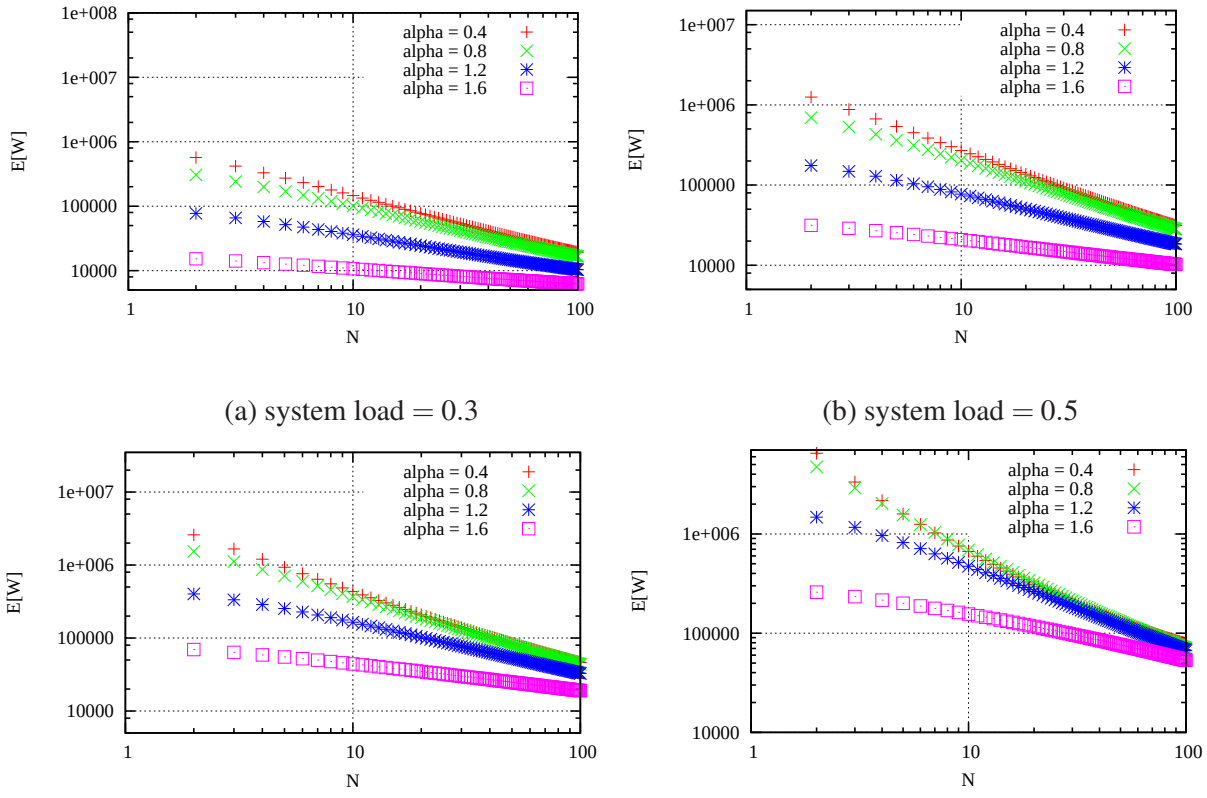
the following manner:

$$E[W]_{(\alpha,load)} = A_w N^{B_w}, \tag{3.22}$$

where N represents the number of levels. Both A_w and B_w are functions of the system load and α. We note that the above equation can be expressed in $y = mx + c$ form as follows:

$$\begin{aligned} E[W] &= A_w N^{B_w}, \\ log(E[W]) &= log(N)B_w + logA_w, \\ y &= mx + c. \end{aligned} \tag{3.23}$$

Figure 3.20 plots the expected waiting vs number of levels for some selected α values and system loads. Note that in these plots x and y axes are in log (base10) scale. The linear relationship between



(a) system load = 0.3

(b) system load = 0.5

(c) system load = 0.7

(d) system load = 0.9

Figure 3.20: Effect of number of levels on the expected waiting time of MLTP-E

$logN$ and $logE[W]$ proves that Equation 3.23 (i.e. power law) is an accurate representation between

the number of levels and the expected waiting time.

Let us now consider a specific example. Let us assume that α is equal to 0.4 and the system load is equal to 0.3. In this particular case we get

$$E[W]_{(0.4,0.3)} = 10^7 N^{-0.883}, \tag{3.24}$$

where N represents the number of levels. We note from Equation 3.24 that as N approaches infinity $E[W]$ approaches zero. However, according to the asymptotic result (refer to [Schrage, 1967]), as N approaches infinity, $E[W]$ should essentially approach a constant value. If we let this constant value be k, we find that k is very small when N lies in the range $(0, 100]$ and therefore, can be discarded.[4]

3.8.2 Expected slowdown for MLTP-E under a large N

Similar to the way it was done for the expected waiting time, we compute the expected slowdown for N-MLTP-E ($0 < N \leq 100$) under various scenarios (i.e. α and system loads) for different N. Then we fit a power curve to each data set and investigate the coefficient of determination. We note that the coefficient of determination (r^2) of these curves are very close to 1. Therefore, the relationship between the expected slowdown and N can be accurately represented using a power law relationship. We obtain

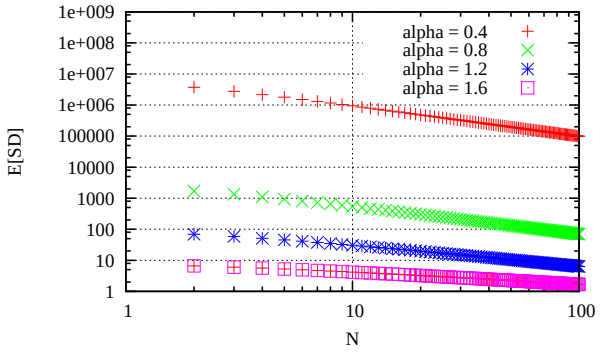
$$\begin{aligned} E[SD] &= A_{sd} N^{B_{sd}}, \\ log(E[W]) &= log(N) B_{sd} + log A_{sd}, \\ y &= mx + c. \end{aligned} \tag{3.25}$$

In the equations above, A_{sd} and B_{sd} are functions of α and the system load. Figure 3.21 plots the expected slowdown vs N for some selected scenarios. Note that both x and y axes in these plots are in log scale. The linear relationship between $log(E[SD])$ and $log(N)$ proves the power law relationship between $E[SD]$ and N.

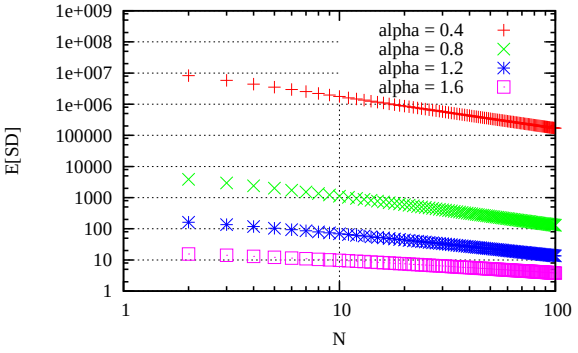
3.9 Performance comparison between N-MLTP-E and FB

Foreground-background (FB) policy can be considered as a coarse-grained approximation for N-MLTP-E when quanta approaches zero and N approaches infinity (refer to Section 2.2.2). Here

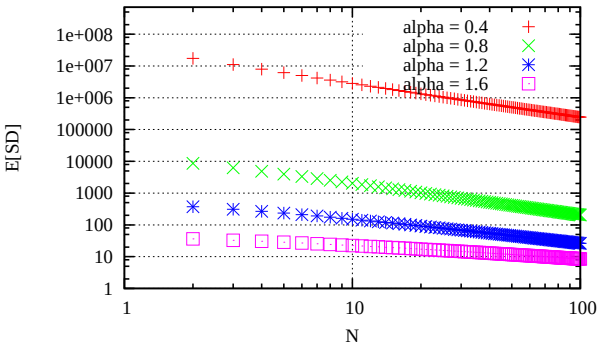
[4]Note that k here is the limiting value of E[W] as N approaches infinity and all quanta approach zero and it is different from the lower bound, k, of the Bounded Pareto distribution.



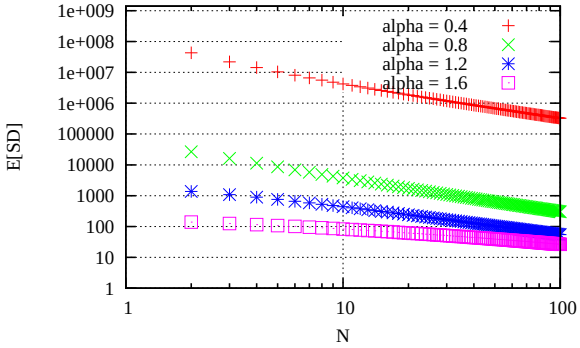
(a) system load = 0.3



(b) system load = 0.5



(a) system load = 0.7



(b) system load = 0.9

Figure 3.21: Effect of number of levels on the expected slowdown of MLTP-E

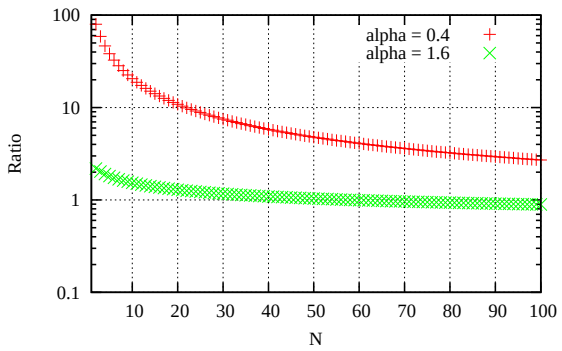
we compare the expected waiting time for N-MLTP-E system with the expected waiting for FB scheduling policy. The aim is to investigate differences in the performance of the two policies.

Let us consider a specific example. Let us for an instance compute the expected waiting time for FB when $\alpha = 0.4$ and the system load is equals to 0.3. The expected waiting time for FB is computed using Equation 2.7 as follows.

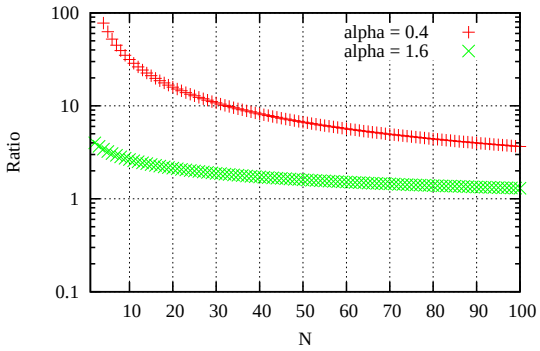
$$\begin{aligned} E(W) &= \int_0^\infty E[W(x)]f(x)dx, \\ &= 4115.54. \end{aligned} \tag{3.26}$$

Note that expected waiting time for FB does not depend on N (number of levels). Figure 3.22 plots the ratio between expected waiting time for N-MLTP-E (up to one hundred levels) and the expected

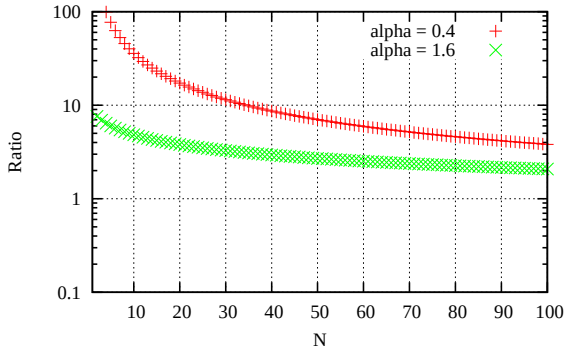
waiting time for FB. We note that as N increases the ratio approaches 1. We also note that under



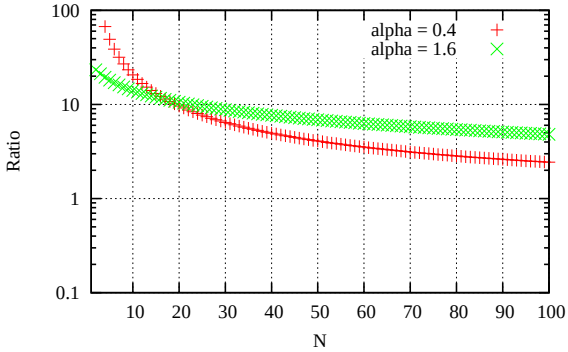
(a) system load = 0.3



(b) system load = 0.5



(a) system load = 0.7

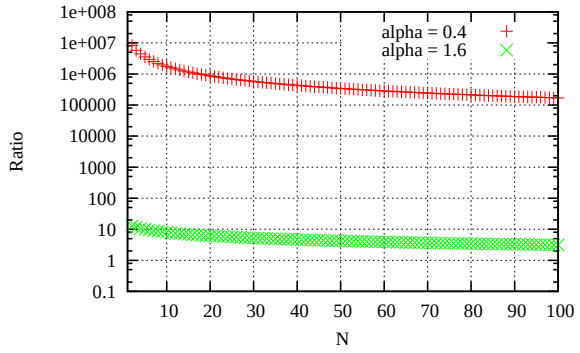


(b) system load = 0.9

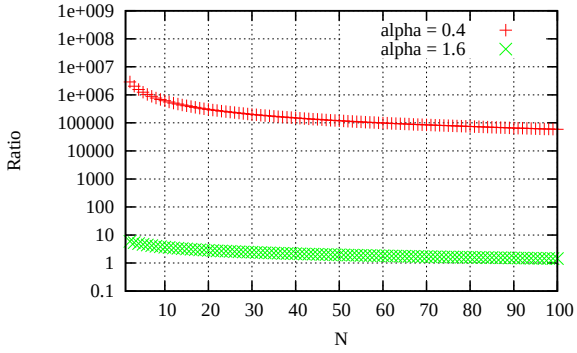
Figure 3.22: Ratio between the expected waiting time of N-MLTP-E and the expected waiting time of FB

highly variable traffic conditions, the ratio is significantly high, which indicates that the expected waiting time for FB is significantly less than that of N-MLTP-E ($0 < N \leq 100$). Therefore, under highly variable traffic conditions, N-MLTP-E requires a large number of levels if it is achieve the same performance levels as FB.

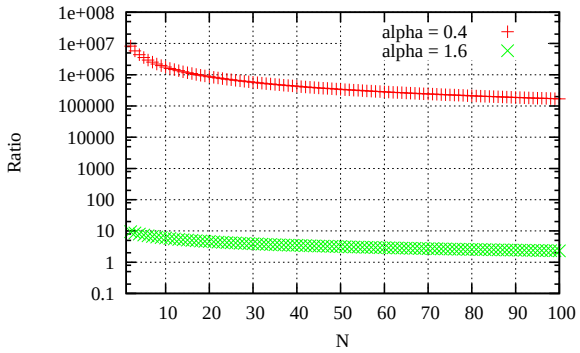
Let us now compare the expected slowdown for N-MLTP-E with the expected slowdown for FB. Figure 3.23 plots the ratio between the expected slowdown for N-MLTP-E and FB. Note that under low α values, the ratio is extremely high even if N is large. This indicates that under highly variable task size distributions, N-MLTP-E requires a large number of queues ($>> 100$) if it is to have the same expected slowdown as FB.



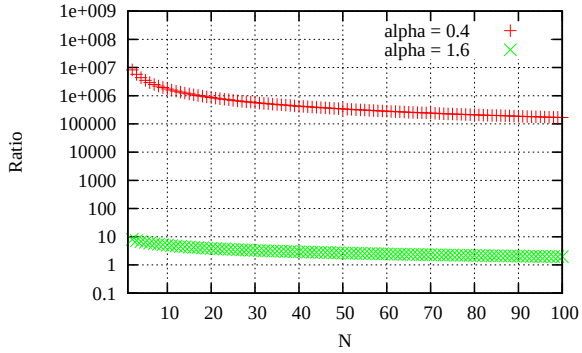
(a) system load = 0.3



(b) system load = 0.5



(a) system load = 0.7



(b) system load = 0.9

Figure 3.23: The ratio between the expected slowdown of N-MLTP-E and the expected slowdown of FB

3.10 Conclusion

In this chapter we investigated the performance of MLTP under heavy-tailed workloads using two performance metrics. While many existing performance evaluations are based on unrealistic assumptions, we investigated performance of MLTP under finite levels when the quanta are not infinitely small. Such a policy is more practical to implement on real computing systems. We showed that N-MLTP-O can result in significant performance improvements over N-MLTP-E and FCFS, especially when both the system load and the task size variabilities are high. We also investigated the load in levels under a range of system loads and task size variabilities and showed that N-MLTP-O uses the technique of unbalancing load to improve the performance. In addition, we investigated the impact of number of levels on the performance and showed that as the number of levels increases the per-

formance improves. We discussed the reasons for such improvements in the performance. We then investigated the impact of quanta on the performance using a 2 level MLTP system and showed that optimal set of quanta are unique for both performance metrics for most of the scenarios. For some workloads, we showed that there exist another set of quanta that would result in near optimal performance. Finally, we investigated the performance of N-MLTP-E under a large number of queues. For both performance metrics, we showed that the relationship between the performance and the number of levels has a power law relationship and the coefficients of the power curve are functions of both the variability of tasks and the system load. We compared the performance of N-MLTP-E with the performance FB and showed that under highly variable traffic conditions, N-MLTP-E requires a large number of queues ($>>$ 100) if N-MLTP-E is to have the same performance levels as FB.

Chapter 4

Performance Modelling and Optimisation in Server Farms

The previous chapter dealt with the ways to optimise the performance in a time sharing server under heavy-tailed service time distributions. The aim of this chapter is to investigate the ways to design efficient task assignment policies for assigning tasks in time sharing server farms under heavy-tailed service time distributions. As discussed in Sections 2.3 and 2.4, size-based task assignment policies [Tari et al., 2005; Harchol-Balter et al., 1999; Harchol-Balter, 2002; Broberg et al., 2004; 2006; Zhang and Sun, 2005; Ciardo et al., 2001] perform well under heavy-tailed workload conditions. The basic idea behind the size-based approach is that each host in the system processes tasks with similar sizes and these size ranges are computed to optimise a given performance criteria (e.g. expected waiting time).

The main limitation of existing size-based policies [Broberg et al., 2004; 2006; Harchol-Balter, 2002; Harchol-Balter et al., 1999] is that these have been designed for batch computing farms and as such, they process tasks using the FCFS scheduling policy. The expected waiting time in a FCFS queue is proportional to the second moment of the service time distribution.[1] Second moment of the heavy-tailed distributions is extremely high.

Although existing size-based policies [Harchol-Balter, 2002; Harchol-Balter et al., 1999; Broberg et al., 2004; 2006] attempt to reduce the variance of task sizes in queues by processing the tasks with similar sizes at the same host, the tasks in server queues can still exhibit high variance in their processing times, especially when the number of hosts in the system is relatively small. As the number of hosts increases, the performance of these size-based policies tends to improve (under

[1] Note that the second moment is closely related to variance.

certain conditions) because a higher number of hosts can achieve higher reduction in the task size variance.

The task assignment policies we propose in this chapter are based on multi-level time sharing policy (MLTP), which we discussed in the previous chapter. The main motivation for using MLTP is its ability to perform well under distributions with the property of decreasing failure rate [Aalto et al., 2004; 2007], a key property of modern heavy-tailed service time distributions.[2] Moreover, MLTP's ability to schedule tasks with unknown service requirements makes it a more attractive policy over policies such as traditional priority queuing models, which assume known processing requirements. Finally, many modern computer systems such as web server systems are inherently time sharing systems making MLTP a more suitable policy for scheduling tasks.

This chapter investigates the performance of three novel task assignment policies, namely, Multi-level Multi-server Task Assignment Policy (MLMS), Multi-level Multi-server Task Assignment Policy based on Task Migration (MLMS-M) and Multi-level Multi-server Task Assignment Policy based on Preemptive Task Migration (MLMS-PM). These policies attempt to improve the performance first by giving preferential treatment to small jobs and second by reducing the variability of task sizes in host queues. MLMS reduces the variability of tasks locally, while MLMS-M and MLMS-PM utilise both local and global variance reduction mechanisms. Both MLMS-M and MLMS-PM facilitate task migration. The key difference between MLMS-PM and MLMS-M is that MLMS-PM facilitates preemptive task migration, whereas MLMS-M does not.

This chapter evaluates the performance of task assignment policies using the most commonly used performance metric, the expected waiting time. Throughout this chapter, it is assumed that the context switch time is negligible and could be equated to zero. In the cases where there are significant context switch overheads, the analytical model presented can be modified to cater for such variations.

The analytical performance analysis of MLMS indicates that MLMS outperforms existing size-based policies under certain conditions. For example, under high system loads and high task size variabilities MLMS with twenty levels outperforms TAGS by a factor of 5. The analytical performance analysis of MLMS-M and MLMS-PM indicates that these policies perform significantly better than existing size-based policies. For example, MLMS-M with five levels outperforms TAGS [Harchol-Balter, 2002] by a factor of 6.75 under highly variable heavy-tailed workloads and high system loads. Under the same conditions, MLMS-PM with five levels outperforms TAGS-PM by a factor of 4. The improvement in the performance depends on the variability of traffic, system load, number of levels and number of hosts. The most significant improvement (in the performance) is

[2]The decreasing failure rate simply means that the longer a task has been processed the less chance it will fail in future.

noticed under very high task size variabilities and high system loads. The performance of MLMS-M improves with the number hosts for certain task size variabilities, while the performance MLMS-PM improves with the number of hosts for all the cases considered.

The rest of this chapter is organised as follows. Sections 4.1, 4.2 and 4.3 present the details of MLMS, MLMS-M and MLMS-PM respectively. The chapter is concluded in Section 4.4.

4.1 Multi-level Multi-server Task Assignment Policy (MLMS)

This section presents the details of MLMS, which is based on MLTP that we discussed in the previous chapter. MLMS is designed for a time sharing server farm that has no task migration facilities. MLMS attempts to improve the performance by reducing the variability of tasks within hosts (i.e. locally) and by giving preferential treatment to small tasks. Figure 4.1 depicts the host architecture for MLMS system. As we note from Figure 4.1, MLMS consists of a central dispatcher (this can be a switch or a router) and n number of back-end hosts that offer mirrored services. The central dispatcher receives new tasks and distributes them among back-end hosts with an equal probability. The back-end hosts process tasks using MLTP, which we discussed in the previous chapter.[3]

4.1.1 Performance model for MLMS

Here we consider the performance analysis of MLMS. The aim is to derive an expression for the expected waiting time of a task under MLMS. For MLMS, the expected waiting time of a task at back-end hosts will be equal because the dispatcher assigns tasks to hosts with an equal probability. Therefore, the (overall) expected waiting time is simply equal to expected waiting time for MLTP in a single host. This follows from the assumption that each host has the same number of levels. The notation given in Table 4.1 is used to describe the MLMS system. Let λ_d and λ be the average arrival rate of tasks at the dispatcher and the average arrival rates of tasks into back-end host respectively. Since the tasks are assigned to hosts with an equal probability, we get

$$\lambda = \frac{\lambda_d}{n}. \tag{4.1}$$

Let Λ_i be the arrival rate into the i^{th} queue of any given host. Then

$$\Lambda_i = (1 - \int_0^{Q_{i-1}} f(x)dx), \tag{4.2}$$

[3]Note that each host under MLMS processes tasks from the full Bounded Pareto distribution (i.e. $B(k, p, \alpha)$), which is not the case for other two policies (MLMS-M and MLMS-PM) that we will discuss next in this chapter

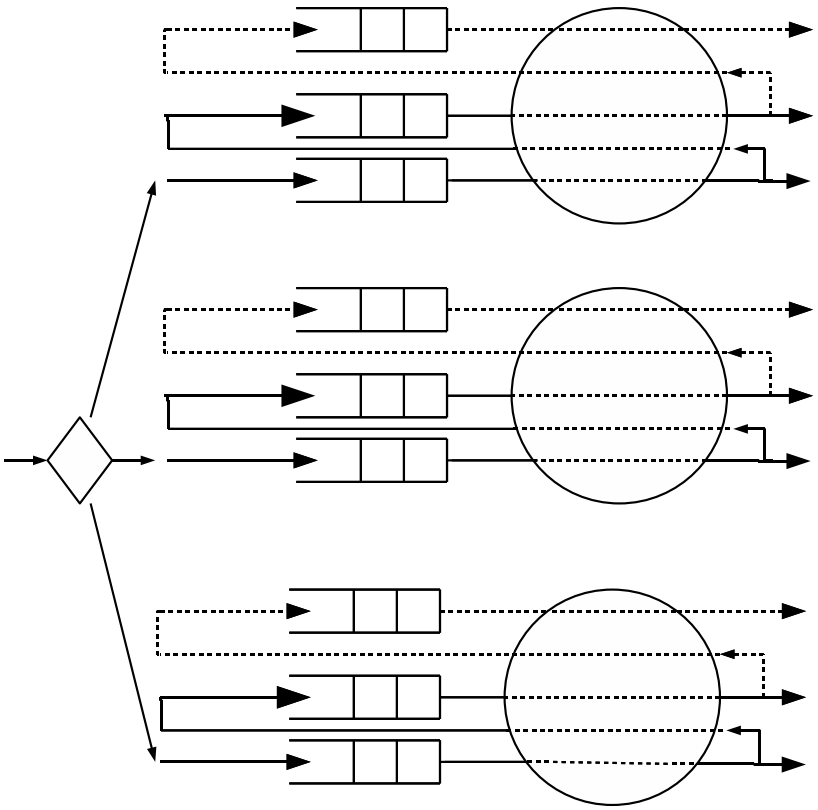


Figure 4.1: MLMS architecture

where $f(x)$ denotes the probability density function of the task size distribution.

The expected waiting time of a task whose size is higher than Q_{i-1} and less than Q_i is given by

$$E[W_i] = \frac{\lambda E[U_i^2] + \sum_{k=i+1}^{N} \Lambda_k E[T_k^2]}{2(1-\lambda E[U_{i-1}])(1-\lambda E[U_i])} + \frac{Q_{i-1}}{(1-\lambda E[U_{i-1}])} - Q_{i-1}, \tag{4.3}$$

where $E[T_i^m]$ and $E[U_i^m]$ are given by

$$E[T_i^m] = \frac{1}{(1-F(Q_{i-1}))}\left(\int_{Q_{i-1}}^{Q_i} (x-Q_{i-1})^m f(x)dx + (Q_i - Q_{i-1})^m (1-F(Q_i))\right), \tag{4.4}$$

$$E[U_i^m] = \int_0^{Q_i} x^m f(x)dx + Q_i^m (1-F(Q_i)). \tag{4.5}$$

N	Number of levels
$B(k,p,\alpha)$	Bounded Pareto (service time) distribution
λ_d	Average outside task arrival rate into system
λ	Average arrival rate into a back-end host
Q_i	i^{th} cut-off point
T_i	The length of processing time that a task in i^{th} queue receives
$E[T_i]$	First moment of the distribution of T_i
$E[T_i^2]$	Second moment of the distribution of T_i
U_i	$T_1+T_2+...+T_i$ if the job returns to the system at least $i-1$ times $T_1+T_2+...+T_k$ if the job returns to the system $k-1$ times, $k<i$
G_i	Probability distribution function of U_i
$E[U_i^k]$	k^{th} moment of G_i
$E[W_i]$	Expected waiting time of a task in i^{th} queue
$E[W]_{MLMS}$	Expected waiting time in the system

Table 4.1: Notation for MLMS

The expected waiting time in the system is obtained by multiplying $E[W_i]$ by the probability that service requirement is within the interval $[Q_{i-1},Q_i], i=1,2,..,N(Q_0=0)$ and then taking the sum of each product term. Let $E[W]_{MLMS}$ the expected waiting time for MLMS. We get

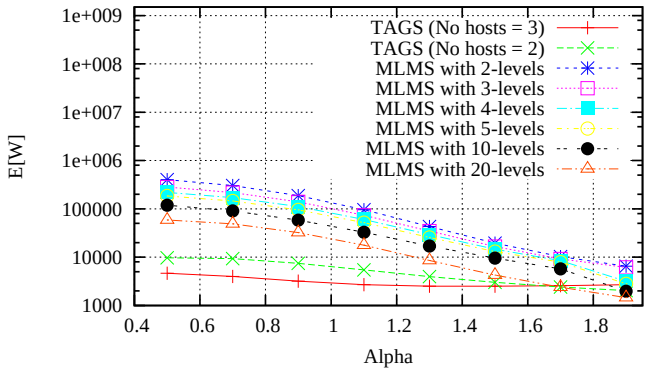
$$E[W]_{MLMS}=\sum_{k=1}^{N}E[W_k]\int_{Q_{k-1}}^{Q_k}f(x)dx. \tag{4.6}$$

When the variability of traffic (α), the system load ($\lambda E[X]$) and number of levels (N) are fixed, $E[W]_{MLMS}$ is a function of $Q_1,Q_2,...,Q_N$, where $k<Q_1<Q_2<...<Q_N=p$. We compute Q_i to optimise the expected waiting time defined by Equation 4.6.

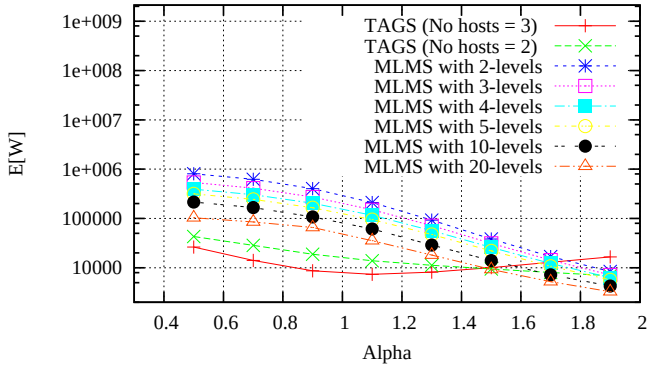
4.1.2 Performance evaluation of MLMS

This section investigates the performance of MLMS by comparing the expected waiting time for MLMS with the expected waiting time for the well known TAGS [Harchol-Balter, 2002]. As discussed in Section 2.4.1, TAGS attempts to improve the performance by minimising the variability of tasks in host queues globally (i.e. at the host level). On the other hand, MLMS attempts to improve the performance by minimising the variability of tasks locally (i.e. within hosts). Both policies give preferential treatment to small tasks. Figure 4.2 illustrates the expected waiting time for MLMS and TAGS in 2 and 3 Host systems respectively.

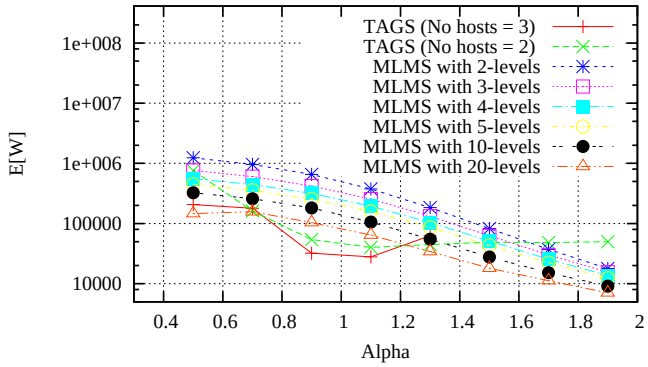
Let us first consider the expected waiting time for two policies (i.e. TAGS and MLMS) in a 2 Host system. We note that under a system load of 0.3, TAGS outperforms MLMS with twenty levels



(a) System load = 0.3



(b) System load = 0.5



(c) System load = 0.7

Figure 4.2: Expected waiting time for MLMS and TAGS in 2 and 3 Host systems

if $\alpha > 1.7$. Similarly, under moderate system loads, TAGS outperforms MLMS with twenty levels if $\alpha > 1.5$. Under low and moderate system loads, we note that MLMS requires a relatively large number of levels if it is to outperform TAGS. For example, under a system load of 0.5, when α is equal to 1.5, MLMS requires twenty levels if it is to have the same expected waiting time as TAGS.

It is worth noting that under a system load of 0.7, MLMS outperforms TAGS in two different α ranges. For example, for a MLMS system with twenty levels, these two ranges are 0.4 - 0.7 and 1.3 - 2.0. We note that under a system load of 0.7, when α = 0.5, MLMS with twenty levels outperforms TAGS by a factor of 5, while under the same system load, when $\alpha = 1.9$, MLMS outperforms TAGS by a factor of 7.

As far as MLMS is concerned, the number of hosts has no impact on the expected waiting time under a given set of conditions because an increase/decrease in the number of hosts does not affect the service time distribution of tasks seen by hosts or the average arrival rates into hosts. On the other hand, the expected waiting time for TAGS varies with the number of hosts under a fixed system load and α, because as the number of hosts increase the average arrival rates and the service time distribution seen by hosts vary.

We note from Figure 4.2 that when α is low, the performance of TAGS is better in a 3 Host system compared to that of a 2 Host system. On the other hand, when α is high, TAGS performs better in a 2 Host system compared to that of a 3 Host system. This means that under low α values, the factor of improvement of TAGS over MLMS is higher in a 3 Host system compared to that of 2 Host system. On the other hand, under (certain) high α values, the factor of improvements of MLMS over TAGS is higher in a 2 Host system compared to that of 3 Host system.

Finally, we note that an increase in the number levels results in an improvement in the performance of MLMS. This is because an increase in the number of levels results in a reduction in the variance of task sizes at individual queues leading to the expected waiting time of small tasks to decrease. This in turn results in an improvement in the overall expected waiting time. Recall that in heavy-tailed distributions, the probability of a small task occurring is very high, while the probability of a very large task occurring is very low.

The advantage of MLMS is that hosts do not need to have task migration facilities. Moreover, MLMS does not kill tasks and restart those from scratch. As such, it does not generate any excess load on the system and therefore, it scales well.

4.2 Multi-level Multi-server Task Assignment Policy based on Task Migration (MLMS-M)

This section provides the details of MLMS-M. MLMS-M is designed for a time sharing server farm that supports non-preemptive migration. MLMS-M improves the performance by minimising the variability of tasks both locally and globally. In addition, it gives preferential treatment to tasks with small processing times. Figure 4.3 depicts MLMS-M. The notation given in Table 4.2 is used to describe MLMS-M.

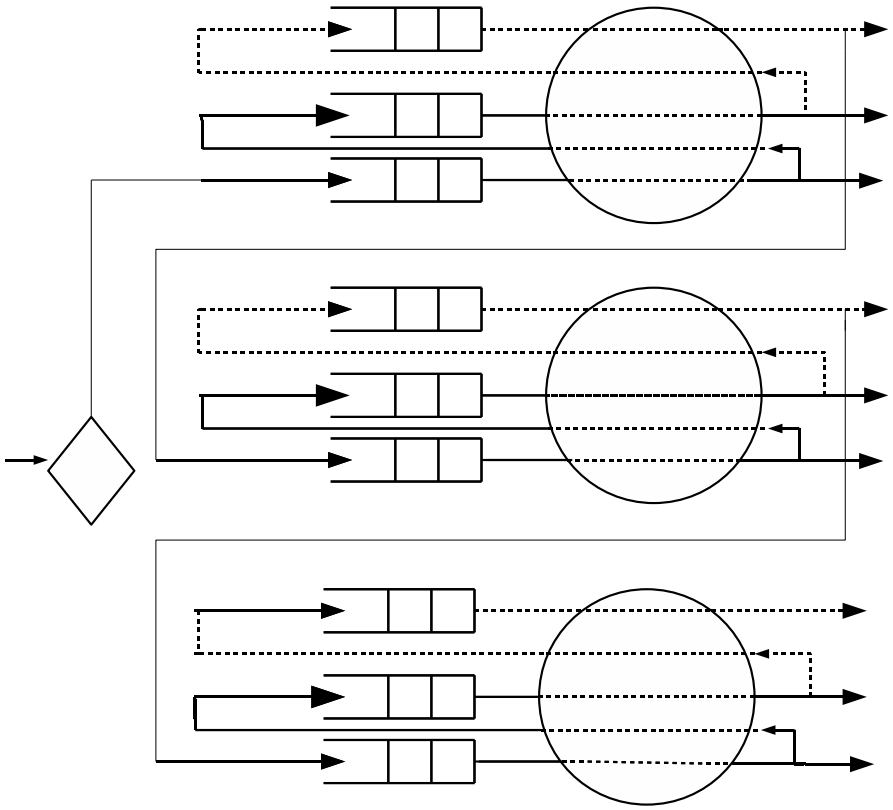


Figure 4.3: Host architecture for MLMS-M and MLMS-PM

The functionality of MLMS-M is described as follows.

- Each task that arrives at the dispatcher is immediately dispatched to Host 1.

n	Number of hosts in the system
N_i	Number of levels at i^{th} Host
$f(x)$	Service time distribution of tasks (i.e. $B(k,p,\alpha)$)
k	Lower bound of the task size distribution
p	Upper bound of the task size distribution
p_i	Fraction of tasks whose final destination is Host i
$p_{(i,j)}$	Fraction of tasks whose final destination is Host i's j^{th} queue
λ	Outside task arrival rate into system
ρ	System load
α	Heavy-tailed parameter
$Q_{(i,j)}$	i^{th} Host's j^{th} cut-off
$T_{(i,j)}$	i^{th} Host's j^{th} simple processing time; the length of processing time that a task in the i^{th} Host's j^{th} queue receives
$E[T_{(i,j)}]$	First moment of the distribution of $T_{(i,j)}$
$E[T^2_{(i,j)}]$	Second moment of the distribution of $T_{(i,j)}$
$U_{(i,j)}$	$T_{(i,1)}+T_{(i,2)}+...+T_{(i,j)}$ if the job returns to the system at least $j-1$ times $T_{(i,1)}+T_{(i,2)}+...+T_{(i,k)}$ if the job returns to the system $k-1$ times, $k<j$
$G_{(i,j)}$	Probability distribution function of $U_{(i,j)}$
$E[U^k_{(i,j)}]$	k^{th} moment of $G_{(i,j)}$
$E[W_i]$	Expected waiting time of a task that spend time at Host i
$E[W]$	Expected waiting time in the system

Table 4.2: Notation for MLMS-M (and MLMS-PM)

- Each task is processed at Host 1 up to $Q(1,N_1)$ using MLTP.
- If the service requirement of a task exceeds $Q(1,N_1)$, the task is killed and migrated to Host 2. Otherwise, the task departs the system.
- Host 2 processes the task in a similar manner from scratch and so on.
- This process continues until the task is fully serviced at which point the task departs the system.

4.2.1 Performance model for MLMS-M

Here we derive an expression for the expected waiting time of a task under MLMS-M. The analysis provided in this section is based on the notation presented in Table 4.2. The objective is to derive an expression for the expected waiting time of MLMS-M.

Let $f(x)$ and $F(x)$ be the probability density function of task sizes and cumulative distribution function of $f(x)$ respectively. Let $p_{(i,j)}$ be the probability that the service time of task is between

$Q_{(i,j-1)}$ and $Q_{(i,j)}$.[4] We obtain

$$p_{(i,j)} = \int_{Q_{(i,j-1)}}^{Q_{(i,j)}} f(x)dx. \tag{4.7}$$

Let p_i be the fraction of tasks whose final destination is Host i

$$p_i = \int_{Q_{(i-1,N_{i-1})}}^{Q_{(i,N_i)}} f(x)dx. \tag{4.8}$$

Let $E[W_{(i,j)}]$ be the expected waiting time of a task in i^{th} Host's j^{th} queue. We obtain

$$E[W_{(i,j)}] = \frac{\lambda_{(i,1)}E[U_{(i,j)}^2] + \sum_{k=j+1}^{N_i} \Lambda_{(i,k)}E[T_{(i,k)}^2]}{2(1-\lambda_{(i,1)}E[U_{(i,j-1)}])(1-\lambda_{(i,1)}E[U_{(i,j)}])} + \frac{Q_{(i,j-1)}}{(1-\lambda_{(i,1)}E[U_{(i,j-1)}])} - Q_{(i,j-1)}. \tag{4.9}$$

In Equation 4.9, $E[T_{(i,j)}]$ represents the expected length of processing time of a task in i^{th} Host's j^{th}. The terms $\lambda_{(i,1)}$ and $\Lambda_{(i,j)}$ in Equation 4.9 denote the arrival rate into the i^{th} Host's 1^{st} queue and the i^{th} Host's j^{th} queue respectively. We obtain these as follows:

$$\lambda_{(i,1)} = \lambda(1 - \int_k^{Q_{(i-1,N_{i-1})}} f(x)dx), \tag{4.10}$$

$$\Lambda_{(i,j)} = \lambda(1 - \int_k^{Q_{(i,j-1)}} f(x)dx) \quad j > 1. \tag{4.11}$$

$E[U_{(i,j)}^m]$ and $E[T_{(i,j)}^m]$ are given by the following equations:

$$E[U_{(i,j)}^m] = \frac{1}{(1-F(Q_{(i-1,N_{i-1})}))}(\int_{Q_{(i-1,N_{i-1})}}^{Q_{(i,j)}} x^m f(x)dx + Q_{(i,j)}^m(1-F(Q_{(i,j)}))), \tag{4.12}$$

$$E[T_{(i,j)}^m] = \frac{1}{(1-F(Q_{(i,j-1)}))}(\int_{Q_{(i,j-1)}}^{Q_{(i,j)}} (x-Q_{(i,j-1)})^m f(x)dx + (Q_{(i,j)}-Q_{(i,j-1)})^m(1-F(Q_{(i,j)}))). \tag{4.13}$$

When we compute $E[U_{(i,j)}^m]$, we have to condition on the tasks with sizes greater than $Q_{(i-1,N_{i-1})}$. Hence, the appearance of the term $\frac{1}{(1-F(Q_{(i-1,N_{i-1})}))}$.

In order to obtain the expected waiting time of a task in the system, we need to first consider the expected waiting time of a task that spends time at Host i. Let $E[W_i]$ be the waiting time of a task at Host i. To obtain $E[W_i]$, we multiply expected waiting time of a task in the i^{th} Host's j^{th} queue (i.e. $E[W_{(i,j)}]$) by the probability that the service time of a task is between $Q_{(i,j-1)}$ and $Q_{(i,j)}$ and we then take the sum of these terms. We have

$$E[W_i] = \sum_{j=1}^{N_i} E[W_{(i,j)}]\frac{p_{(i,j)}}{p_i}. \tag{4.14}$$

We can now write an expression for the expected waiting time of a task in the system (i.e. overall expected waiting time). Let n be the number of hosts in system and let $E[W]_{MLMS-M}$ be the expected

[4]Note that the sizes of tasks processed in Queue i are in the range $[Q_{(i,j-1)}, Q_{(i,j)}]$.

waiting time in the system. $E[W]_{MLMS-M}$ is given by

$$E[W]_{MLMS-M} = E[W_1]p_1 + (E[W_1] + E[W_2])p_2 + ... + (E[W_1] + ... + E[W_n])p_n. \tag{4.15}$$

When the variability of traffic, system load and number of levels are fixed, $E[W]_{MLMS-M}$ is a function of $Q_{(1,1)}, Q_{(1,2)}, ..., Q_{(n,N_n)}$, where $k < Q_{(1,1)} < Q_{(1,2)} < ... < Q_{(n,N_n)} = p$. We compute these cut-offs to optimise the expected waiting time.

Recall that MLMS-M restarts certain tasks from scratch and as a result, it generates some excess load in the system. Let L_i be the load on Host i. Then,

$$L_i = \lambda_{(i,1)} E[U_i], \tag{4.16}$$

where $E[U_i]$ is given by

$$E[U_i] = \frac{\int_{Q_{(i-1,N_{i-1})}}^{Q_{(i,N_i)}} xf(x)dx}{(1 - F(Q_{(i-1,N_{i-1})}))} + \frac{(Q_{(i,N_i)})(1 - F(Q_{(i,N_i)}))}{(1 - F(Q_{(i-1,N_{i-1})}))} E[U_i], \tag{4.17}$$

and $\lambda_{(i,1)}$ denotes the average arrival rate of tasks into the Host i's Queue 1 (refer to Equation 4.10). Let L_{sum} be the sum of individual host loads. Then,

$$L_{sum} = \lambda_{(1,1)} E[U_1] + \lambda_{(2,1)} E[U_2] + .. + \lambda_{(n,1)} E[U_n]. \tag{4.18}$$

Let L_{excess} be the excess load on the system. Then,

$$L_{excess} = L_{sum} - \lambda E[X], \tag{4.19}$$

where λ and $E[X]$ denote the average arrival rate into the system and the mean of the service time distribution respectively. $\lambda E[X]$ can be referred as the desired sum of loads in the system and $\frac{\lambda E[X]}{n}$ is called the system load.

4.2.2 Performance analysis of MLMS-M

This section provides the performance analysis of MLMS-M. In section 4.2.2.1 we investigate the expected waiting time for MLMS-M under low, moderate and high system loads. Sections 4.2.2.2 and 4.2.2.3 investigate the effect of queue arrangement on the expected waiting time and the impact of hosts on the expected waiting time respectively.

4.2.2.1 Performance evaluation of MLMS-M for the case of 2 hosts

Here we investigate the expected waiting time for MLMS-M in a 2 Host system using Equation 4.15, which we derived in Section 4.2.1. Figure 4.4 shows the expected waiting time for MLMS-

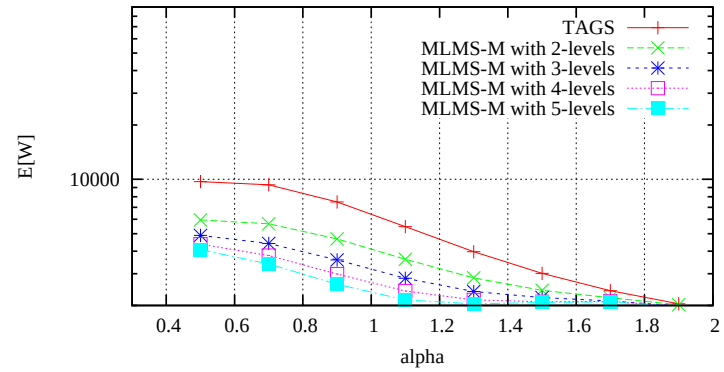
M. We note that MLMS-M outperforms TAGS for almost all the cases considered. The highest improvement in the performance is seen under high system loads and very high task sizes variabilities (i.e. low α values). For example, under a system load of 0.7, when $\alpha = 0.5$, MLMS-M with two levels outperforms TAGS by a factor of 2.7. Under the same conditions, MLMS-M with five levels outperforms TAGS by a factor 6.75.

Figure 4.5 shows the effect of number of levels on the expected waiting time for MLMS-M. We note that there is an improvement in the expected waiting time with the number of levels. We also note that the rate at which the performance improves, decreases with the number of levels. Note that as the number of levels increases, the rate at which the performance improves decreases. For example, under a system load of 0.7, when $\alpha = 0.5$, MLMS-M with three levels outperforms MLMS-M with two levels by a factor of 1.5, while under the same conditions MLMS-M with four levels outperforms MLMS-M with three levels only by a factor of 1.3.

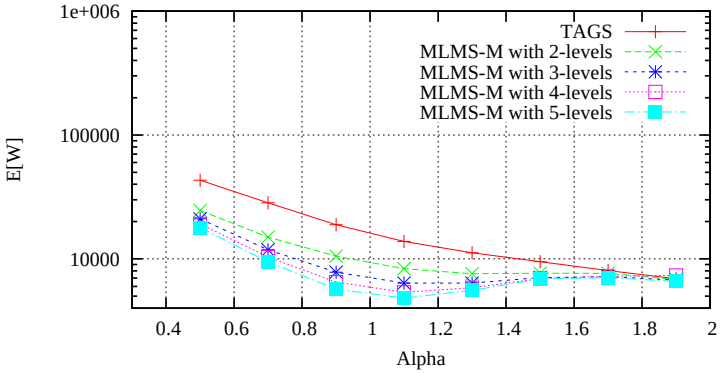
We note from Figure 4.4 that under a fixed system load the expected waiting time for MLMS-M does not continuously decrease with increasing α. This is particularly the case under moderate and high system loads when the number of levels is high. As α increases, the expected waiting time decreases up to a minimum value and then it begins to increase. Recall that MLMS-M restarts certain tasks from scratch at (destination) hosts, which results in an additional load (excess load) on the system. The behaviour of excess load with α is illustrated in Figure 4.6. We note that excess load decreases up to a minimum value and then it begins to increase. The excess load under MLMS-M is high under high α, because under high α values, MLMS-M migrates a large number of tasks to Host 2, to ensure that the load on both hosts are less than 1. This means that large number of tasks are killed and restarted from scratch under high α values resulting in large amounts of excess. We find that for a given number of levels and a system load there may exist unique α that produces the best waiting time. For example, under a system load of 0.7, when the number of levels is equal to 5, this optimal value of α lies in the range 1 and 1.2. We also see from Figure 4.6 that the number of levels has no effect on the excess load if the system load is a constant. If we can configure MLMS-M to support preemptive migration (this type of policies do not restart jobs from scratch rather resume their execution) the performance will continuously improve with increasing α, provided that the cost of migration is negligible. This we will consider later in chapter.

4.2.2.2 Effect of queue arrangement for the case of 2 hosts

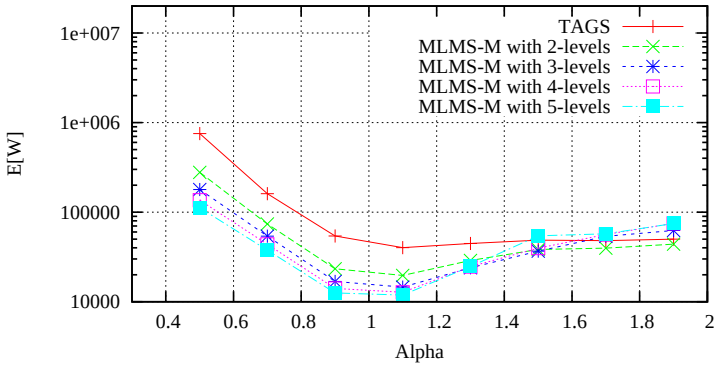
By analysing the effect of queue arrangement on the expected waiting time, we can determine the optimal configuration (i.e. optimal number of queues) for (individual) hosts under specific workload



(a) System load = 0.3

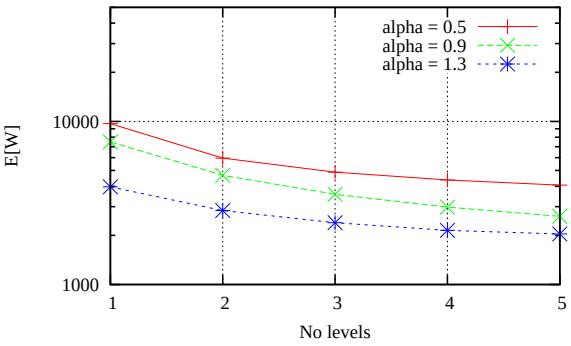


(b) System load = 0.5

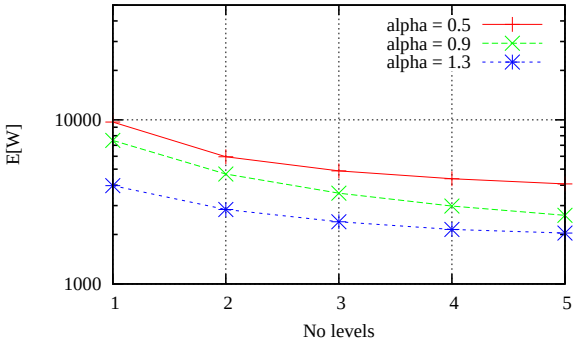


(c) System load = 0.7

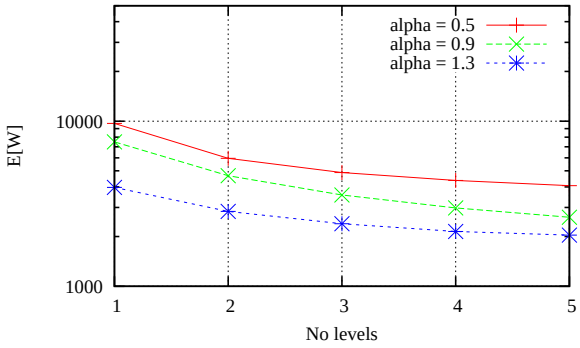
Figure 4.4: Expected waiting time for MLMS-M and TAGS in a 2 *Host system*



(a) System load = 0.3



(b) System load = 0.5



(c) System load = 0.7

Figure 4.5: Effect of levels on the expected waiting time of MLMS-M in a 2 Host system

scenarios. In previous sections we assumed that all hosts in the server farm have equal number of queues. Here we investigate the effect of number of queues on the expected waiting time when the total number of queues in the system is equal to four. We note that there are three distinct ways to have four queues in a 2 Host system:

- A-1: Host 1 has two queues and Host 2 has two queues.
- A-2: Host 1 has one queue and Host 2 has three queues.
- A-3: Host 1 has three queues and Host 2 has one queue.

The expected waiting time for above systems are computed by optimising the corresponding expected waiting time expressions. Figure 4.7 shows the expected waiting time for A-1, A-2 and A-3. Under low (0.3) and moderate (0.5) system loads, A-2 outperforms both A-1 and A-3. This means that under these system loads, reducing the variance of task sizes in host queues at Host 2 is more important than

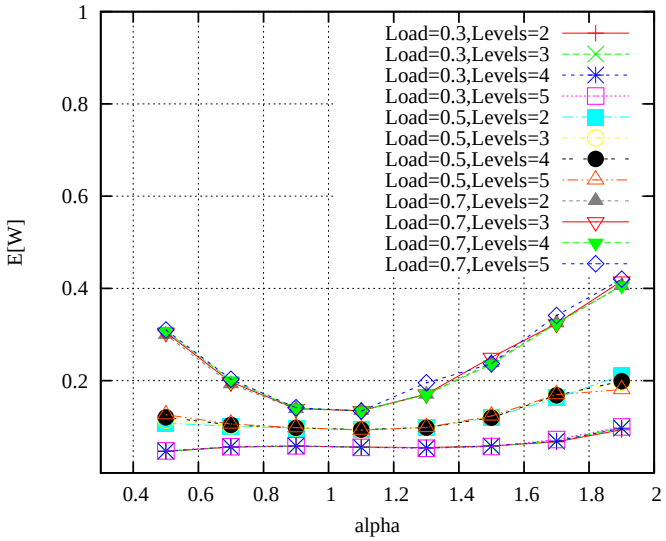


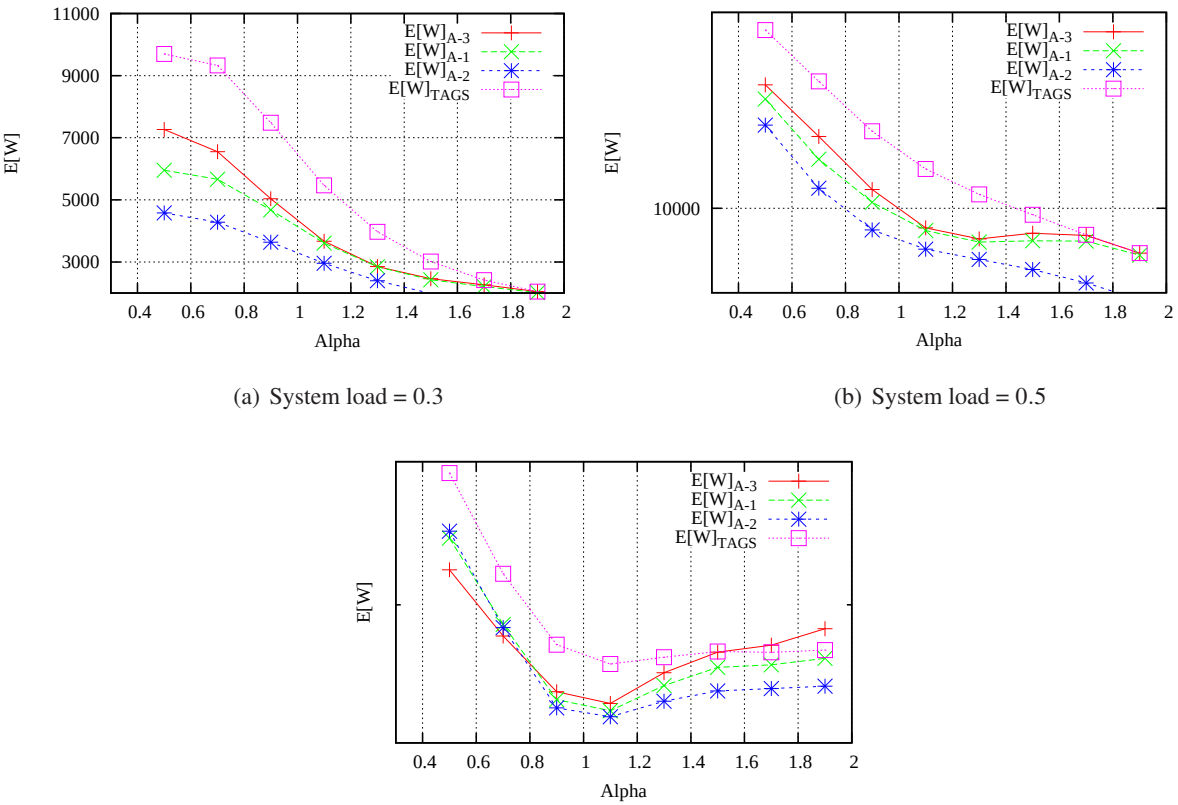
Figure 4.6: Excess load under MLMS-M in a 2 Host system

that of Host 1. Note that under high system loads A-2 outperforms other policies only if α is high. If α is low (and the system load is high), A-3 has the best expected waiting time. Let $E[W]_{A-1}$, $E[W]_{A-2}$ and $E[W]_{A-3}$ be the expected waiting time under A-1, A-2 and A-3 respectively. The following table provides the summary of results.

$\rho = 0.3$	$E[W]_{A-3} > E[W]_{A-1} > E[W]_{A-2}$
$\rho = 0.5$	$E[W]_{A-3} > E[W]_{A-1} > E[W]_{A-2}$
$\rho = 0.7$, $\alpha > 0.9$	$E[W]_{A-3} > E[W]_{A-1} > E[W]_{A-2}$
$\rho = 0.7$, $\alpha < 0.9$	$E[W]_{A-2} > E[W]_{A-1} > E[W]_{A-3}$

4.2.2.3 Effect of number of hosts

This section investigates the effect of number of hosts on the expected waiting time for MLMS-M. This analysis is important as it allows us to get an idea of the effect of number of hosts on the performance of MLMS-M. Here we compare the expected waiting time for MLMS-M in a 2 Host system with that of a 3 Host system. Figure 4.8 shows these results. We note that as the number of hosts increases, the expected waiting time for MLMS-M increases under certain α values. For example, under a system load of 0.5, when $\alpha = 0.9$, MLMS-M with three levels performs 1.5 times better in a 3 Host system compared to that of a 2 Host system. If α is greater than a certain value, an increase in the number of hosts no longer results in an improvement in the expected waiting time.



(a) System load = 0.3

(b) System load = 0.5

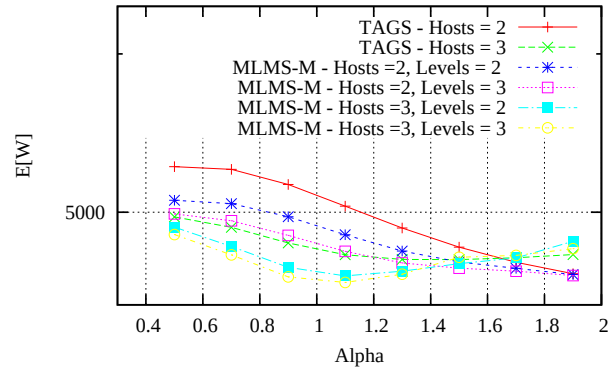
(c) System load = 0.7

Figure 4.7: Effect of queue arrangement on the expected waiting time of MLMS-M in a 2 Host system

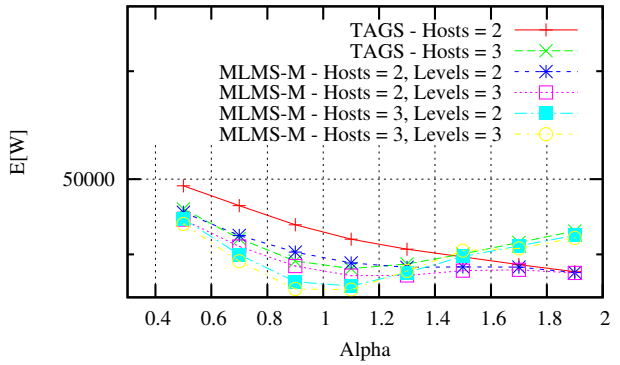
For example, under a system load of 0.5, when $\alpha > 1.3$, MLMS-M with three levels performs better in a 2 Host system compared to that of a 3 Host system.[5] Although a 3 Host system offers higher reduction in the variance of task sizes, MLMS-M generates higher amounts of excess load under high α values. This makes the performance of MLMS-M in a 3 Host system to deteriorate if α is greater than a specific threshold value. There are two ways to address this issue.

- Preemptive migration: Under preemptive migration tasks are not restarted from scratch at their destination hosts, rather their execution is resumed. This we consider in the next section.
- Devise a new model that minimises the excess load for high α values. This we will consider in the next chapter.

[5] We noted in Section 2.4, the performance of advanced task assignment policies (e.g. TAGS) do not improve with the number of hosts under all conditions.



(a) System load = 0.3



(b) System load = 0.5

Figure 4.8: Impact of hosts on the expected waiting time for MLMS-M

4.3 Multi-level Multi-server Task Assignment Policy based on Preemptive Migration (MLMS-PM)

MLMS-PM is the last model we consider in this chapter and it is designed for a server farms that supports preemptive migration. MLMS-PM aims to address the following three main issues associated with MLMS-M.

- MLMS-M restarts tasks from scratch, which results in a significant amount of wasted processing.
- MLMS-M does not scale well under certain workload scenarios (e.g. high system loads and high α). This is because wasted processing generated under MLMS-M due to restarting tasks

from scratch results in the expected waiting time to increase.

- The expected waiting time for MLMS-M deteriorates significantly if the processing time variability decreases beyond certain a threshold.

MLMS-PM addresses these issues by facilitating preemptive migration (work-conserving migration) of tasks between hosts. The key features of MLMS-PM are as follows.

- It gives preferential treatment to task with short processing requirements.[6]
- It utilises a 2-level variance reduction mechanism.[7]
- It allows preemptive migration (work-conserving migration) of tasks between hosts.[8]

The host architecture for MLMS-PM is similar to the host architecture of MLMS-M. Note that each host in MLMS-M has a designated time limit associated with it, where the designated time limit of Host i is given by $Q_{(i,N_i)}$. The functionality of MLMS-PM can be explained in the following manner.

- Whenever a new task arrives at the dispatcher, it is immediately dispatched to Host 1.
- Host 1 processes the task according to the multi-level time sharing policy (MLTP) described in Chapter 3.
- If the service requirement of the task exceeds the designated time limit assigned to Host 1, i.e. $Q_{(1,N_1)}$, the task is migrated to the next host. Otherwise, the task departs the system from Host 1.
- Host 2 resumes the execution of the task and processes it in a similar manner using the MLTP and so on.
- This process continues until the task is fully serviced, at which point the task departs the system.

In this chapter we focus on negligible cost preemptive migration, where the tasks possess minimal state information. It is possible for certain types of tasks to incur significant amounts of migration cost on the system. Two types of cost-based migration scenarios can be defined: fixed cost migration and proportional cost migration. Under fix cost migration, a fix migration cost incurs when a task is

[6] This is property of both MLMS and MLMS-M.
[7] This is property of MLMS-M.
[8] MLMS and MLMS-M do not have this property.

migrated from one host to another, whereas under the proportional cost migration, the migration cost is proportional to the processing requirement of the task being migrated. We will consider these two migration criteria in the next chapter.

As was assumed for MLMS and MLMS-M, we assume that the arrival process into the system is Poisson. This is a reasonable assumption for many types of computer workloads [Cao et al., 2001]. Moreover, for modelling purposes, we assume that the arrival process from Host i to Host $i+1$ is also Poisson. Note that much previous work is based on this assumption [Harchol-Balter, 2002; Broberg et al., 2004; 2006].

4.3.1 Performance model for MLMS-PM

This section provides the performance model for MLMS-PM. This model is somewhat similar to the performance model for MLMS-M. However, there are some important differences due to the fact that MLMS-PM is based on preemptive migration. We use the notation given in Table 4.2 to describe MLMS-PM. Our aim is to derive an expression for the expected waiting time in the system. In order to derive an expression for the expected waiting time we need to first derive an expression for $E[W_i]$, the expected time of a task that spends time at Host i. $E[W_i]$ is computed by multiplying the expected waiting time of a task in i^{th} Host's j^{th} queue (i.e. $E[W_{(i,j)}]$) by the probability that the service time of a task is between $Q_{(i,j-1)}$ and $Q_{(i,j)}$. (i.e. the fraction of tasks whose final destination is Host i's j^{th} queue.)

Let $p_{(i,j)}$ be the fraction of tasks whose final destination is Host i's j^{th} queue. We obtain

$$p_{(i,j)} = \int_{Q_{(i,j-1)}}^{Q_{(i,j)}} f(x)dx. \tag{4.20}$$

Let p_i be the fraction of tasks whose final destination is Host i. We obtain

$$p_i = \int_{Q_{(i-1,N_{i-1})}}^{Q_{(i,N_i)}} f(x)dx. \tag{4.21}$$

Let $E[W_{(i,j)}]$ be the expected waiting time of a task in i^{th} Host's j^{th} queue. We obtain

$$E[W_{(i,j)}] = \frac{\lambda_{(i,1)}E[U^2_{(i,j)}] + \sum_{k=j+1}^{N_i} \Lambda_{(i,k)}E[T^2_{(i,k)}]}{2(1-\lambda_{(i,1)}E[U_{(i,j-1)}])(1-\lambda_{(i,1)}E[U_{(i,j)}])} + \frac{Q_{(i,j-1)}}{(1-\lambda_{(i,1)}E[U_{(i,j-1)}])} - Q_{i,j-1}. \tag{4.22}$$

Let us now discuss how to compute each term in Equation 4.22. Note that $Q_{(i,j-1)}$ and $E[T_{(i,j)}]$ denote the total processing time up to i^{th} host's $j-1$ level and the expected processing time of a task at i^{th} host's j^{th} level.

$E[T^m_{(i,j)}]$ given by

$$E[T^m_{(i,j)}] = \frac{1}{(1-F(Q_{(i,j-1)}))}(\int_{Q_{(i,j-1)}}^{Q_{(i,j)}} (x-Q_{(i,j-1)})^m f(x)dx + (Q_{(i,j)}-Q_{(i,j-1)})^m(1-F(Q_{(i,j)}))). \tag{4.23}$$

$E[U_i^m]$ is given by

$$E[U^m_{(i,j)}] = \frac{1}{(1-F(Q_{(i-1,N_{i-1})}))}(\int_{Q_{(i-1,N_{i-1})}}^{Q_{(i,j)}} (x-Q_{(i-1,N_{i-1})})^m f(x)dx + (Q_{(i,j)}-Q_{(i-1,N_{i-1})})^m(1-F(Q_{(i,j)}))). \tag{4.24}$$

$E[U_i^m]$ conditions on the distribution of task's remaining size by considering the work already done, i.e. $Q_{(i-1,N_{i-1})}$.

$\lambda_{(i,1)}$ and $\Lambda_{(i,j)}$ (in Equation 4.22) denote the arrival rate into i^{th} Host's 1^{st} queue and i^{th} Host's j^{th} queue respectively. We obtain these as follows:

$$\lambda_{(i,1)} = \lambda(1-\int_k^{Q_{(i-1,N_{i-1})}} f(x)dx), \tag{4.25}$$

$$\Lambda_{(i,j)} = \lambda(1-\int_k^{Q_{(i,j-1)}} f(x)dx), \quad j>1. \tag{4.26}$$

Let $E[W_i]$ be the waiting time of a task at Host i. We have

$$E[W_i] = \sum_{j=1}^{N_i} E[W_{(i,j)}]\frac{p_{(i,j)}}{p_i}. \tag{4.27}$$

We can now write an expression for the expected waiting time of a task in the system (i.e. the overall expected waiting time). Let n be the number of hosts in system and let $E[W]$ be the expected waiting time. $E[W]$ is given by

$$E[W] = E[W_1]p_1 + (E[W_1]+E[W_2])p_2 + ... + (E[W_1]+...+E[W_n])p_n. \tag{4.28}$$

In the above equation, we have conditioned on the final destination of tasks. When the variability of processing requirements, the system load and the number of levels are fixed, $E[W]$ (Equation 4.28) is a function of $Q_{(1,1)}, Q_{(1,2)}, ..., Q_{(n,N_n)}$, where $k < Q_{(1,1)} < Q_{(1,2)} < ... < Q_{(n,N_n)} = p$. We compute these cut-offs so as to optimise the expected waiting time.

4.3.2 Performance evaluation of MLMS-PM

This section provides a detailed performance analysis of MLMS-PM under a range of task size variabilities and system loads. Section 4.3.2.1 provides the analytical performance analysis of MLMS-PM for the case of two hosts. In Section 4.3.2.2 we investigate the effect of levels on the performance of MLMS-PM. Section 4.3.2.3 compares the performance of MLMS with MLMS-PM. Performance of MLMS-PM for the case of more than 3 hosts is investigated in Section 4.3.2.4. Finally, in Section

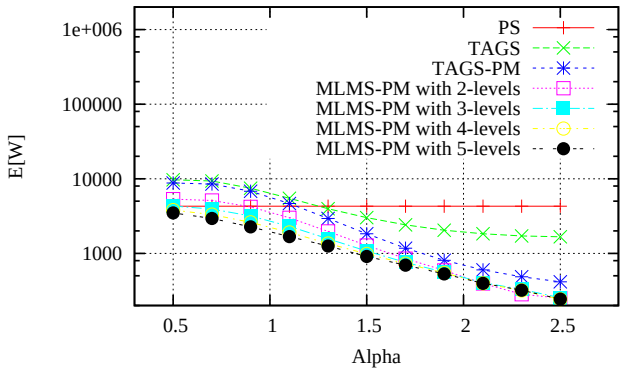
4.3.2.5 we investigate the effect of queue arrangement on the performance.

4.3.2.1 Performance evaluation of MLMS-PM for the case of 2 hosts

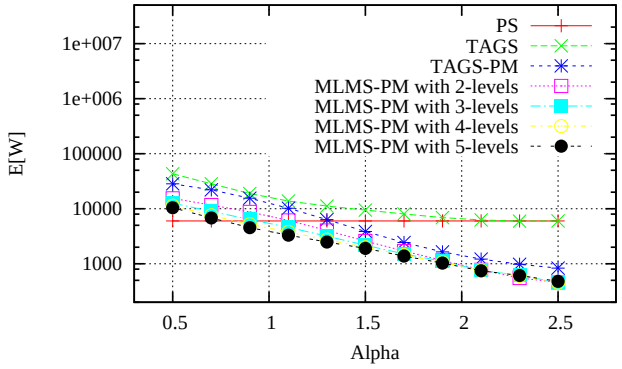
This section evaluates the performance of the proposed policy against the performance of four other core task assignment policies, namely, TAGS, TAGS-PM, PS and MLMS. More details about TAGS, TAGS-PM and PS can be found in Chapter 2. It is worth noting that we do not compare the performance of MLMS-PM with the performance of Foreground-Background policy (FB) [Nuyens and Wierman, 2008].[9] Figure 4.9 depicts the results under the system loads of 0.3 (low), 0.5 (moderate) and 0.7 (high). In Figure 4.9 we have excluded MLMS policy and we will consider this later in this chapter. It is worth noting that all the expected waiting time plots are presented on a log scale for the y-axis. As expected, the expected waiting time of all policies degrade as the system load increases. We note that under a fixed system load, the expected waiting time for policies tend to improve with increasing α. However, there is one exception to this, i.e. TAGS. The behaviour of TAGS is previously discussed in Section 2.4.1. We note that the expected waiting time for PS does not depend on α and it is only a function of the average arrival rate and system load. We also note that under low system loads (0.3), TAGS outperforms PS under a range of α values. However, under moderate and high system loads, TAGS does not outperform the PS. TAGS-PM, on the other hand, outperforms PS under a range of α values. We note that the expected waiting time curve for PS intersects the expected waiting time curve for TAGS-PM when α is in the range 1 and 1.5. As the system load increases, there is a slight movement of this intersection point to the right direction.

Let us now consider the expected waiting time of a task under MLMS-PM. First note that it does not suffer from the problem that TAGS suffered, i.e. its performance does not degrade after certain α. Second we note that MLMS-PM outperforms PS under a wide range α values. For example, under a system load of 0.5, when α equals 1.1, MLMS-PM with five levels outperforms PS by a factor of 2, while under the same system load, when α equals 2.5, MLMS-PM with five levels outperforms PS by a factor of 12. On the other hand, PS outperforms MLMS-PM if α is very small. Third we note that MLMS-PM outperforms TAGS-PM for all the scenarios considered. The highest improvement in the performance is obtained under high system loads and low α values. For example, under a system

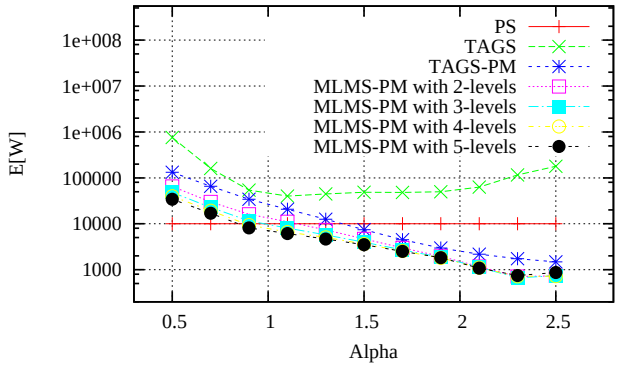
[9] As was pointed out in Section 2.2.2, FB gives priority to the task that has so far received the least amount of service. If there are n such tasks, then they are serviced simultaneously. Under a FB system, whenever a new task arrives at the system, the task currently being serviced is preempted from service and the new task is processed as much as the service of the task/tasks that was/were preempted. FB has two issues. First it is inefficient because each time a new task arrives, the task currently being processed has to be preempted. Under high arrival rates this process can be very inefficient and costly. Second FB is rather difficult implement in a typical time sharing system. Time sharing systems are quantum-based systems that process tasks up to a fixed amount of time (called the quantum).



(a) System load = 0.3



(b) System load = 0.5



(c) System load = 0.7

Figure 4.9: Expected waiting time for MLMS-PM and TAGS in a 2 Host system

load of 0.7, when α equals 0.5, MLMS-PM with five levels outperforms TAGS-PM by a factor of 4. Under low α values, MLMS-PM performs significantly better than TAGS-PM even under low system loads. For example, under a system load of 0.3, when α equals 0.5, MLMS-PM outperforms TAGS-PM by a factor of 2.3. The factor of improvement in the expected waiting time for the MLMS-PM over TAGS-PM is minimal if α is close to 2.5.

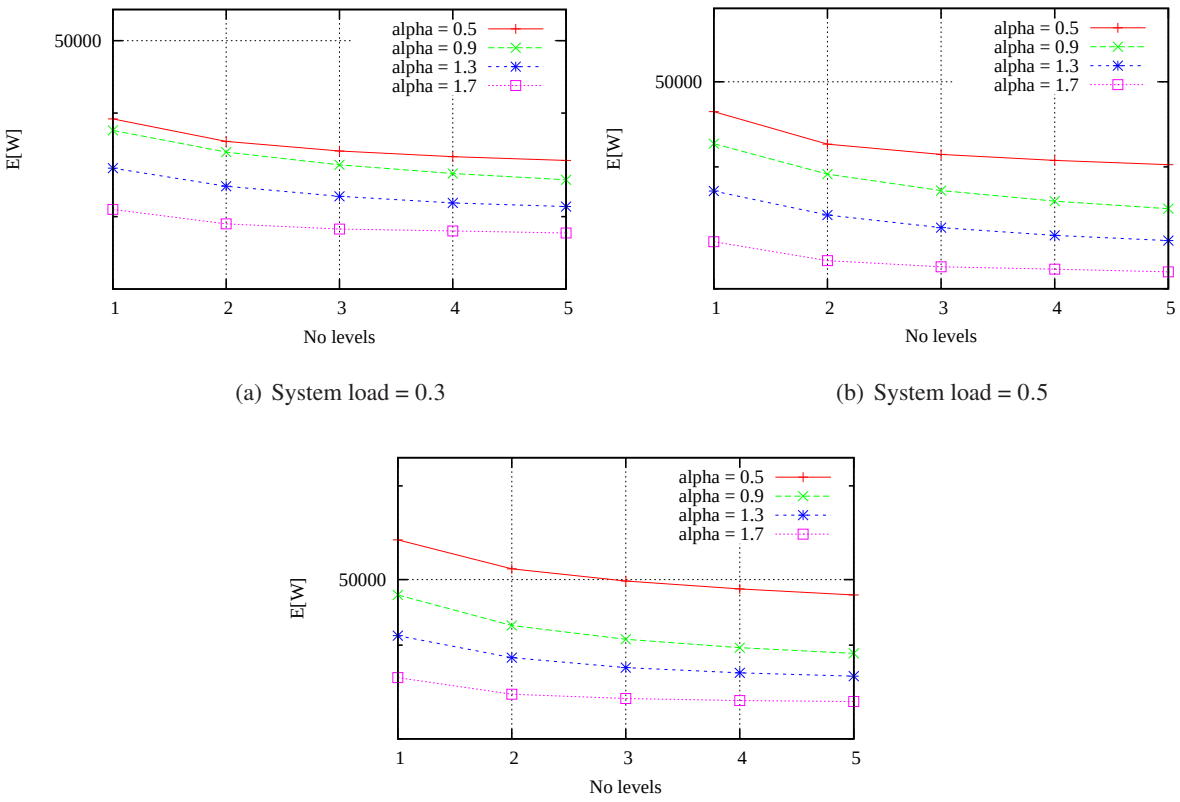
4.3.2.2 Effect of the number of levels on the performance of MLMS-PM

In the case of MLMS and MLMS-M an increase in the number of levels results in the expected waiting time to decrease. This section investigates the effect of the number of levels on the expected waiting time for MLMS-PM. The aim is to investigate under which workload conditions we can get significant performance improvements by increasing the number of levels. Figure 4.10 depicts the effect of the number of levels on the expected waiting time. We note that the performance of MLMS-PM improves with the number of levels. For example, under a system load of 0.5, when α is equal to 0.5, MLMS-PM with five levels outperforms MLMS-PM with two levels by a factor of 1.5. As we increase the number of levels, the task size variability decreases in host queues. This results in an improvement in the expected waiting time for tasks with relatively small processing requirements leading to the improvement in the overall performance. However, the improvement in the performance is only significant between lower number of levels.

4.3.2.3 Performance comparison of MLMS with MLMS-PM

This section compares the expected waiting time for MLMS with that of MLMS-PM. Recall that both MLMS and MLMS-PM process tasks according to MLTP at individual hosts. However, MLMS-PM supports preemptive task migration between hosts, whereas MLMS does not. Figure 4.11 illustrates the expected waiting time for MLMS (up to ten levels) and MLMS-PM (up to five levels).

We note that MLMS-PM performs significantly better than MLMS under all the system loads considered. Even with a large number of levels, MLMS does not outperform MLMS-PM. For example, under a system load of 0.5, when $\alpha = 0.5$, MLMS-PM with two levels outperforms MLMS with ten levels by a factor of 24. Under the same conditions, MLMS-PM with four levels outperforms MLMS with ten levels by a factor of 42. Clearly, the 2-level variance reduction (global and local) model has resulted in significant performance improvements.



(a) System load = 0.3

(b) System load = 0.5

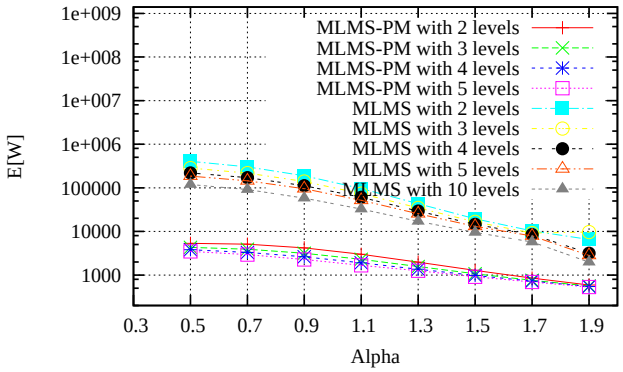
(c) System load = 0.7

Figure 4.10: Effect of number of levels on the expected waiting time for MLMS-PM in a 2 Host system

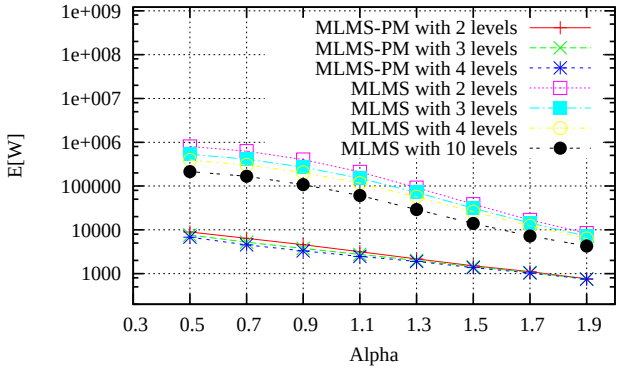
4.3.2.4 Expected waiting time for MLMS-PM for the case of more than 2 hosts

This section investigates the expected waiting time for MLMS-PM when the number of servers in the server farm is greater than 2. The objective is to investigate under which workload conditions MLMS-PM are more suitable for assigning tasks in large-sized server farms.

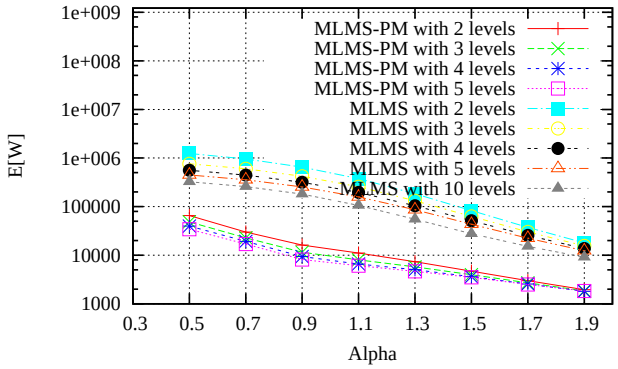
Let us first compare the expected waiting time for MLMS-PM in a 2 Host system with that of a 3 Host system. Figure 4.12 plots the expected waiting time for MLMS-PM with five levels in 2 and 3 Host systems. We note that under a fixed system load, MLMS-PM performs better in a 3 Host system compared to that of a 2 Host system under all the scenarios considered. The reason for this is as follows. As the number of hosts increases from 2 to 3, both $E[W_1]$ (expected waiting time of a task that spends time at Host 1) and $E[W_2]$ (expected waiting time of a task that spends time at Host 2) tend to decrease because the variability of processing times decreases at Host 1 and Host 2. This means that tasks with relatively short processing times are processed faster in a 3 Host system compared to



(a) System load = 0.3

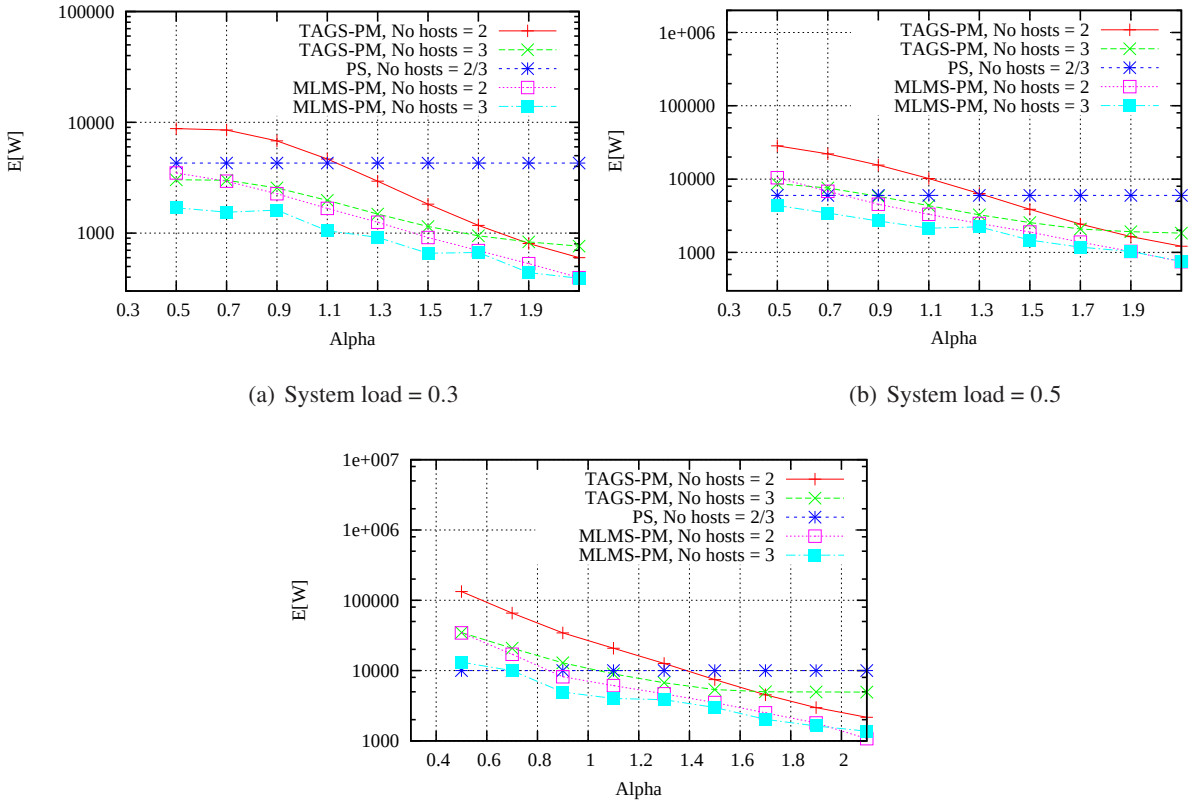


(b) System load = 0.5



(c) System load = 0.7

Figure 4.11: Expected waiting time for MLMS and MLMS-PM in a 2 Host system



(a) System load = 0.3

(b) System load = 0.5

(c) System load = 0.7

Figure 4.12: Expected waiting time for MLMS-PM (with five levels), TAGS-PM and PS in 2 and 3 Host systems

that of a 2 Host system. Therefore, in a 3 Host system there is an improvement in the waiting time of short tasks (compared to that of a 2 Host system), which in turn results in an improvement in the overall expected waiting time. The highest improvement in the expected waiting time is seen under low α values and high system loads. For example, under a system load of 0.7, when α equals 0.5, MLMS-PM performs 2.5 times better in a 3 Host system compared to that of a 2 Host system. Under low and moderate system loads, there is still a significant improvement in the performance if α is not very high.

In section 4.3.2.1 we noted that in a 2 Host system, PS outperforms MLMS-PM under a system loads of 0.5 and 0.7, when α is very low. In the case of a 3 Host system, MLMS-PM outperforms PS even under very low α values. For example, under a system load of 0.5, when α equals 0.5, MLMS-PM with five levels outperforms PS by a factor of 1.3. The magnitude of this performance

improvement grows as α approaches 2.5.

Let us now briefly investigate the performance of MLMS-PM when the number of hosts in the system is greater than 3. Figure 4.13 illustrates the performance of MLMS-PM under a system load of 0.5 in 2, 3 and 4 Host systems. Clearly, an increase in the number of hosts results in an improvement

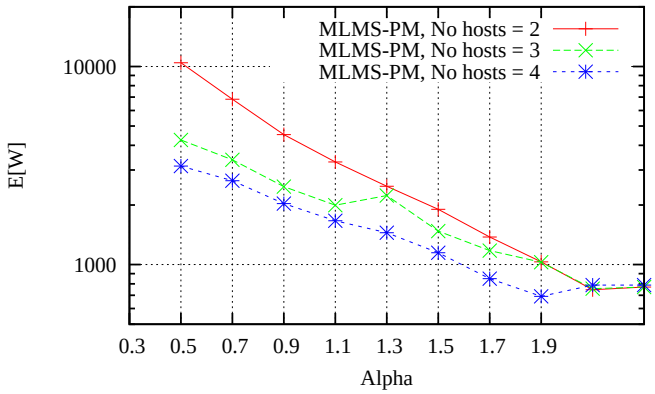


Figure 4.13: Effect of number of hosts on the expected waiting time for MLMS-PM: system load = 0.5

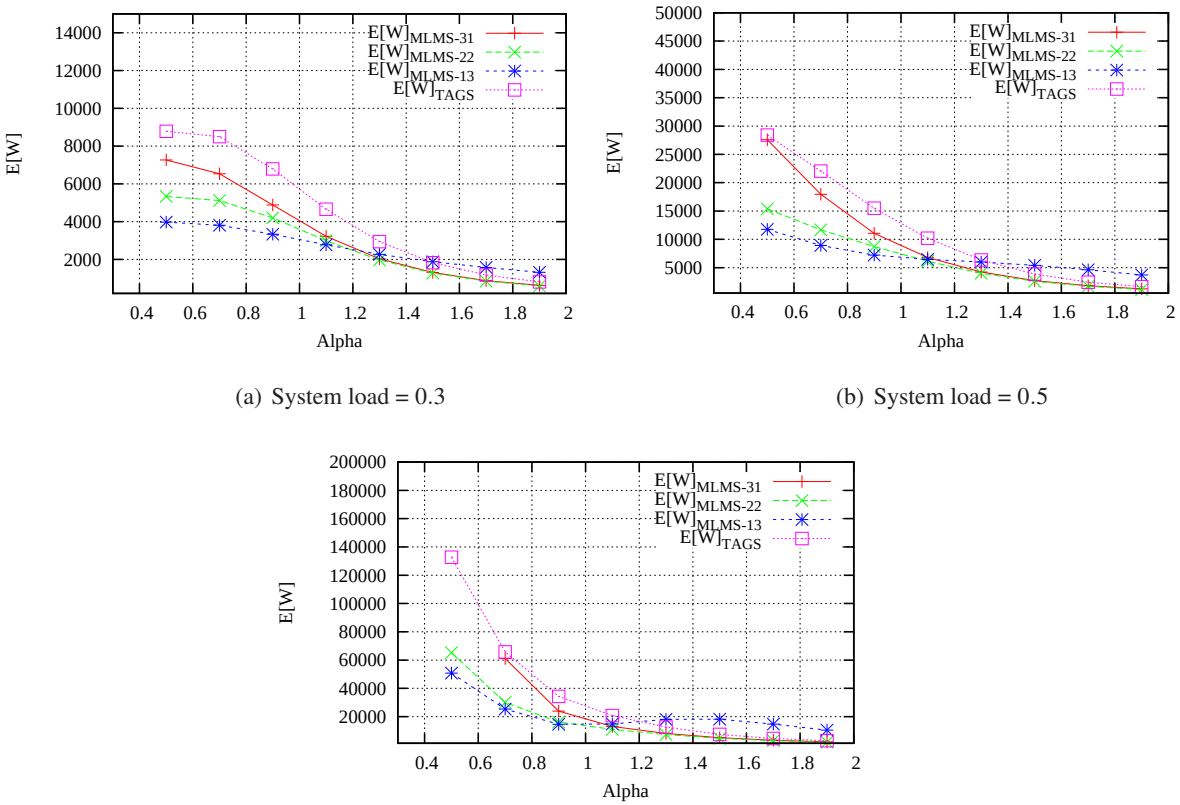
in the expected waiting time. The highest improvement in the performance is seen under low α values. We note that as α increases, the improvement in the expected waiting time decreases. We also note that the rate at which expected waiting time improves, decreases with the number of hosts. For example, under a system load of 0.5, when α equals 0.5, MLMS-PM with five levels performs 2.5 times better in a 3 Host system compared to that of a 2 Host system. Under the same conditions, MLMS-PM with five levels performs only 1.5 times better in a 4 Host system compared to that of a 3 Host system.

4.3.2.5 Effect of queue arrangement

Similar to the analysis provided in Section 4.2.2.2 for MLMS-M, this section investigates the effect of queue arrangement on the expected waiting time for MLMS-PM. The objective is to determine the optimal number of queues for individual hosts under different workload scenarios.

Let us assume that the number of queues is equal to 4. As pointed out, there are three distinct ways to place four queues in a 2 Host system. Let these be A-1, A-2 and A-3, where A-1 consists of two queues on each host, A-2 consists of one queue on Host 1 and three queues on Host 2 and A-3 has three queues on Host 1 and one queue on Host 2. The expected waiting time for each of these

models are computed by optimising the corresponding performance metrics. Figure 4.14 illustrates the expected waiting time for three models. We see that A-2 outperforms the other two models under



(a) System load = 0.3

(b) System load = 0.5

(c) System load = 0.7

Figure 4.14: Effect of queue arrangement on the expected waiting time for MLMS-PM in a 2 Host system

high task size variabilities ($\alpha < 1.1$). For example, under a system load of 0.7, when α is low, A-2 outperforms A-1 and A-3 by 28% and 250% respectively. Under low task size variabilities ($\alpha < 1.1$), A-1 and A-3 tend to perform (slightly) better than A-2.

4.4 Conclusion

This chapter proposed three novel task assignment policies all of which are based on MLTP. MLTP was used as the underlying scheduling policy due to two reasons. First it has the ability improve the performance under distributions with the property of decreasing failure rate. Second it can schedule tasks with unknown processing requirements. The analysis provided in this chapter was based on

heavy-tailed service time distributions because heavy-tailed distributions have been proven to represent many realistic computer workloads. We investigated three models, namely, MLMS, MLMS-M and MLMS-PM. MLMS is the simplest model, which assigns tasks to hosts with an equal probability. While MLMS does not facilitate task migration, both MLMS-M and MLMS-PM supports task migration between hosts. The key difference between MLMS-M and MLMS-PM is that MLMS-PM supports preemptive migration, whereas MLMS-M only supports non-preemptive migration.

The analytical performance analysis of policies indicates that MLMS outperforms TAGS under certain scenarios, while MLMS-M outperforms TAGS for all the scenarios considered. Similarly, MLMS-PM outperforms TAGS-PM for all the scenarios considered. The most significant performance improvement is seen under high task size variabilities and high system loads. The factor of improvement is dependent on the variability of traffic, system load, number of levels and the number of hosts. When all other factors are constant, an increase in the number of levels results in an improvement in the performance (for all policies considered). However, the rate of improvement decreases with the number of levels. In the case of MLMS-M, an increase in the number of hosts does not always improve the performance. However, for MLMS-PM the performance always improves with the number of hosts.

Chapter 5

Task Assignment in Multiple Server Farms using Preemptive Migration and Flow Control

The task assignment policies proposed in the previous chapter were designed for stand-alone server farms. This chapter investigates the way to design efficient task assignment policies to assign tasks in multiple server farms. Existing task assignment policies [Harchol-Balter, 2002; Broberg et al., 2004; 2006; Ciardo et al., 2001; Harchol-Balter et al., 1999; Tari et al., 2005; Fu and Tari, 2003] are not very efficient in assigning tasks in multiple server farms because they have not been designed to exploit the properties of such environments. With the availability of high speed networks (e.g. a fibre optics network can provide maximum data transfer rates of more than 100 Gbps) and operating systems that have features such as preemptive migration, there exists a window of opportunity to design more efficient task assignment policies for assigning tasks in multiple server farms. Such policies can better utilise the resources in multiple server farm environments and therefore, can perform better compared to those that optimise the performance in stand-alone server farms. The previous chapter dealt with task assignment in time sharing server farms. This chapter concentrates on batch computing server farms. The main reason for considering batch server farms is that advanced task assignment policies, which exploit the properties of multiple server farm environments can be more beneficial for batch server farms compared to that of time sharing server farms. This chapter has made two contributions.

1. We propose an efficient task assignment policy for a stand-alone batch server farm. This policy is called Multi-tier Task Assignment with Minimum Excess Load (MTTMEL) and it addresses the core limitations of existing traditional task assignment policies (e.g. poor performance

under high task size variabilities) and the main limitations of well known TAGS (e.g. poor performance under low and moderate task size variabilities, poor performance under high system loads and poor performance in large-sized server farms). MTTMEL is based on a flexible multi-tier host architecture, where the hosts in tiers only process tasks whose sizes are within a certain size range. By grouping and processing tasks in such a manner, MTTMEL reduces the variance of tasks in hosts queues, which leads to significant performance improvements.[1] This multi-tier host architecture of MTTMEL offers a high degree of flexibility in terms of how many tiers and hosts to be used in server farms. These parameters can be computed to optimise the performance under a given scenario. Furthermore, this multi-tier host architecture speeds up the flow of small tasks by processing these small tasks in a relatively large set of hosts. This minimises the expected waiting time of small tasks, which leads to an overall improvement in the performance.[2]

2. The main (second) contribution of this chapter is a novel task assignment policy for assigning tasks in multiple server farms. We propose Multi-cluster Task Assignment based on Preemptive Migration (MCTPM). MCTPM is based on the same multi-tier host architecture, which we discussed earlier in (1). MCTPM controls the traffic flow into server farms via a global dispatching device so as to optimise the performance. MCTPM supports preemptive task migration between servers in the same farm and between servers in different farms. We provide an analytical model for the proposed policy taking into account the cost of migration. We then carry out an extensive analytical and experimental performance analysis of the proposed policy under a wide range of workload conditions. Detailed description of how the policy works can be found in Section 5.2. The core features of MCTPM are as follows.

   - Multi-tier host architecture: This host architecture is identical to the host architecture of MTTMEL.
   - Flow control: MCTPM controls the traffic flow into server farms using a global dispatching device, where the incoming tasks first arrive. The aim of this flow control model is twofold: 1) it ensures that no server farm gets overloaded under very high arrival rates and 2) it ensures that the correct fractions of tasks are assigned to server farms so as to optimise the performance. These fractions are computed based on the properties of incoming traffic (i.e. average arrival rate and service time distribution) and the properties

[1] The expected waiting time/expected slowdown of a task in a first-come-first-served (FCFS) queue is proportional to the variance of the service time distribution [Kleinrock, 1975].

[2] The probability of small task occurring is very high under heavy-tailed distributions.

of server farms (i.e. number of tiers and number hosts in tiers). By having such control over the traffic flow, MCTPM manages to achieve significant performance improvements over existing policies under a wide range of workload scenarios. To support flow control, MCTPM is featured with several task dispatching devices. These dispatching devices control the traffic flow into server farms and tiers.

- Preemptive migration: Preemptive migration ensures that MCTPM can resume the execution of a task that was previously suspended at a different host. Although preemptive migration can be expensive, we show that the proposed policy performs well even under very high migration costs. It is worth pointing out that in the previous chapter, we have already considered preemptive migration. The main difference between the work presented in this chapter and the previous chapter is that in the previous chapter we assumed that migration cost is negligible, whereas in this chapter we assume that migration cost is not zero (this is typically the case for tasks processed in batch computing environments).

Both MTTMEL and MCTPM are especially designed for tasks that exhibit high variability in their service times because there is extensive evidence indicating that service time distributions of tasks in computing environments are best represented by heavy-tailed distributions (refer to Section 2.1.2 for more information about workload properties).

Performance analysis of MTTMEL shows that it outperforms existing policies under a wide range of workload conditions. For example, it outperforms Random by a factor of 23 and TAGS by a factor of 2.6 under certain scenarios. Performance analysis of MCTPM shows that it outperforms both traditional and recent models significantly under a wide range of workload conditions. For example, it outperforms existing models such as MC-Random, MC-TAGSPM and MC-MTTPM by factors of 190, 5 and 10.5 respectively under specific conditions. These scenarios will be discussed later in this thesis.

The rest of this chapter is structured as follows. Section 5.1 presents MTTMEL and its performance analysis. Section 5.2 presents the main contribution of this chapter, i.e. MCTPM and its performance analysis. The chapter is concluded in Section 5.3.

5.1 Multi-tier Task Assignment with Minimum Excess Load (MTTMEL)

This section presents a novel task assignment policy called MTTMEL, for a (batch) server farm that consists of a central dispatcher and a set of homogeneous back-end hosts that offer mirrored services. MCTPM, which is presented in Section 5.2, is based on this task assignment policy. MTTMEL has

been designed to address the following two main issues associated with the well known TAGS.

- TAGS performance degrades significantly under low and moderate task size variabilities because it generates large amounts of excess load under these conditions.
- TAGS does not scale well, particularly under high system loads because it generates large amounts of excess load under high system loads.

To address the above two problems, MTTMEL is featured with a multi-tier host architecture. This novel host architecture minimises excess load on the system by minimising the number times tasks that are killed and restarted from scratch. As such, MTTMEL performs significantly better than TAGS, particularly under low and moderate task size variabilities. In addition, it scales well under all system loads (i.e. low, moderate and high system loads).

5.1.1 Overview of MTTMEL

This section provides an overview of MTTMEL. The host architectures for MTTMEL for the case of three and five hosts are illustrated in Figures 5.1 and 5.2 respectively. We note that MTTMEL

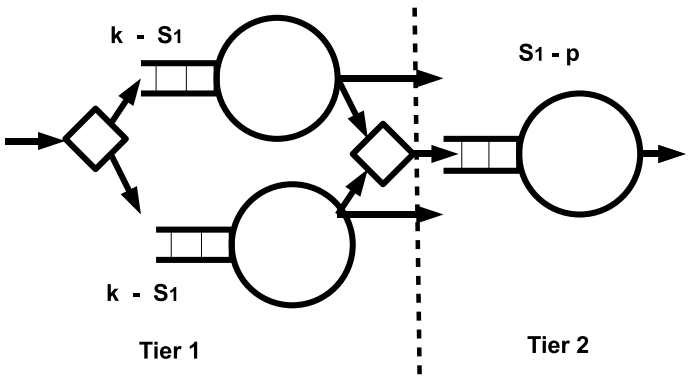


Figure 5.1: MTTMEL system with three hosts

consists of multiple host tiers, where each tier has a task size range associated with it. In each tier,

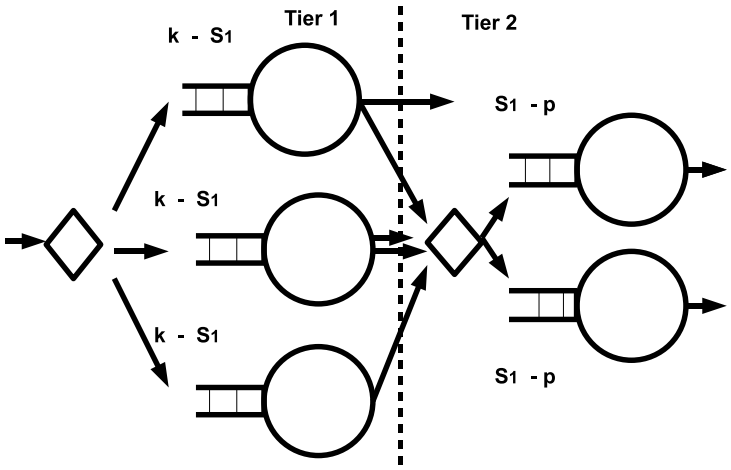


Figure 5.2: MTTMEL system with five hosts

tasks are processed up to the upper limit of the task size range. Let $[s_{i-1}, s_i]$ be the task size range assigned to Tier i. Then, hosts in Tier i process tasks only up to s_i.

The basic functionality of MTTMEL is as follows. Each task arriving at the central dispatcher is assigned to a host in Tier 1 with an equal probability. The task is processed up to the upper limit of Tier 1's task size range (i.e. s_1) in a FCFS manner. If the service time of the task is greater than the upper limit of Tier 1, the task is killed and redirected to a host in Tier 2 via an intermediate dispatching device with an equal probability.[3] Otherwise the task departs system. If the task arrives at Tier 2, it is restarted from scratch at a Tier 2 host and processed in a similar manner and so on.[4] This process continues until the task is fully serviced at which point the task departs the system. The time limits for tiers are computed to optimise the expected waiting time. The number of hosts in Tier 1 is greater than the number of hosts in Tier 2 and the number of hosts in Tier 2 is greater than the number of hosts in Tier 3 and so on. The aim is to process tasks with relatively short processing requirements faster by assigning these tasks to many hosts. Another possible host arrangement for a 5 Host system is four hosts in Tier 1 and one host in Tier 2. The number of tiers that would result in the best performance will depend on the factors such as variability of traffic and arrival rate into the system.

[3]Note that the average arrival rates into hosts in a particular tier are the same as the tasks are dispatched to tiers with an equal probability.

[4]Note that similar to TAGS, MTTMEL is based on non-preemptive migration.

5.1.2 Performance model for MTTMEL

The performance model for MTTMEL is presented in this section. The main objective is to derive an expression for the expected waiting time. This will be used later to evaluate the performance of MTTMEL.

Let $[s_{i-1}, s_i]$ be the task size range assigned to Tier i and p_i be the probability that the service time of a task is between s_{i-1} and s_i. p_i is given by

$$p_i = \int_{s_{i-1}}^{s_i} f(x)dx, \tag{5.1}$$

where $f(x)$ is the service time distribution of tasks.

Let $E[W_{if}]$ be the expected waiting time of a task whose final destination is a host in Tier i. $E[W_{if}]$ is given by

$$E[W_{if}] = \sum_{j=1}^{i} E[W_{jv}], \tag{5.2}$$

where $E[W_{jv}]$ denotes the expected waiting time of a task that visits a host in Tier j. $E[W_{jv}]$ is obtained using the Pollaczek-Khinchin formula [Kleinrock, 1975]

$$E[W_{jv}] = \frac{\lambda_j E[X_{jv}^2]}{2(1-\lambda_j E[X_{jv}])}, \tag{5.3}$$

where $E[X_{jv}]$ and $E[X_{jv}^2]$ denote the 1^{st} and 2^{nd} moments of the processing time distribution of tasks that visit a host in Tier j. λ_j denotes the arrival rate into a Tier j host and T denotes the number of tiers in the system. Later in this chapter, we will discuss how to compute λ_j for distributed systems with different number of hosts. $E[X_{jv}^l]$ can be obtained using the following formula.

$$E[X_{jv}^l] = \frac{p_j}{\sum_{k=j}^{T} p_k} E[X_j^l] + \frac{\sum_{k=j+1}^{T} p_k}{\sum_{k=j}^{T} p_k} s_j^l, \tag{5.4}$$

where $E[X_j^l]$ denotes the l^{th} moment of the distribution of tasks whose final destination is a host in Tier j. $E[X_j^l]$ is given by

$$E[X_j^l] = \int_{s_{j-1}}^{s_j} \frac{1}{p_j} x^l f(x)dx. \tag{5.5}$$

Finally, the expected waiting time in the system is computed as

$$E[W] = \sum_{i=1}^{T} E[W_{if}]p_i. \tag{5.6}$$

Note that the expected waiting time of a task is the same for all hosts in a given tier since all hosts in the tier receive tasks from the same distribution and with the same arrival rate. $E[W]$ is a function of α, $s_1, s_2, .., s_{n-1}$ and λ_i. For a given set of α and λ_i, we compute $s_1, s_2, .., s_{n-1}$ so as to optimise the expected waiting time.

Note that for a MTTMEL system consisting of n number of hosts, the arrival rate at the dispatcher is given by

$$\lambda = \frac{\rho * n}{E[X]}, \tag{5.7}$$

where ρ is the system load and $E[X]$ is the expected task size.

5.1.3 Performance evaluation of MTTMEL

The analytical performance analysis of MTTMEL in 3 and 6 Host systems is presented in this section. The expected waiting time for MTTMEL is compared against the expected waiting time of two well known policies namely, Random [Silberschatz et al., 1998] and TAGS [Harchol-Balter, 2002].

Let us first consider expected waiting time for MTTMEL in a 3 Host system with two tiers. For a 3 Host system (illustrated in Figure 5.1), λ_1 (the average arrival rate into Tier 1) and λ_2 (the average arrival rate into Tier 2) are computed as follows:

$$\begin{aligned} \lambda_1 &= \frac{\lambda}{2}, \\ \lambda_2 &= \lambda p_2, \end{aligned} \tag{5.8}$$

where λ denotes the average arrival rate into the system (given by Equation 5.7) and p_i is given by Equation 5.1. Figure 5.3 illustrates the expected waiting time for MTTMEL in a 3 Host system. We note that MTTMEL outperforms Random under a wide range of α values and the factor of improvement is highly significant when α lies in the range 0.7-1.5. For example, under a system load of 0.5, when α equals 1.1, MTTMEL outperforms Random by a factor of 23. In addition, we note that MTTMEL outperforms TAGS under a range of α. For example, under a system load of 0.7, when α equals 1.3, MTTMEL outperforms TAGS by a factor of 2.6. Note that under high system loads (i.e. 0.7), TAGS fails to operate in steady state, if α is greater than 1.3. This means that we cannot find cut-offs for hosts (task size ranges) such that load on each host is less than 1. MTTMEL, however, does not suffer from this problem and it can function in steady state throughout the entire

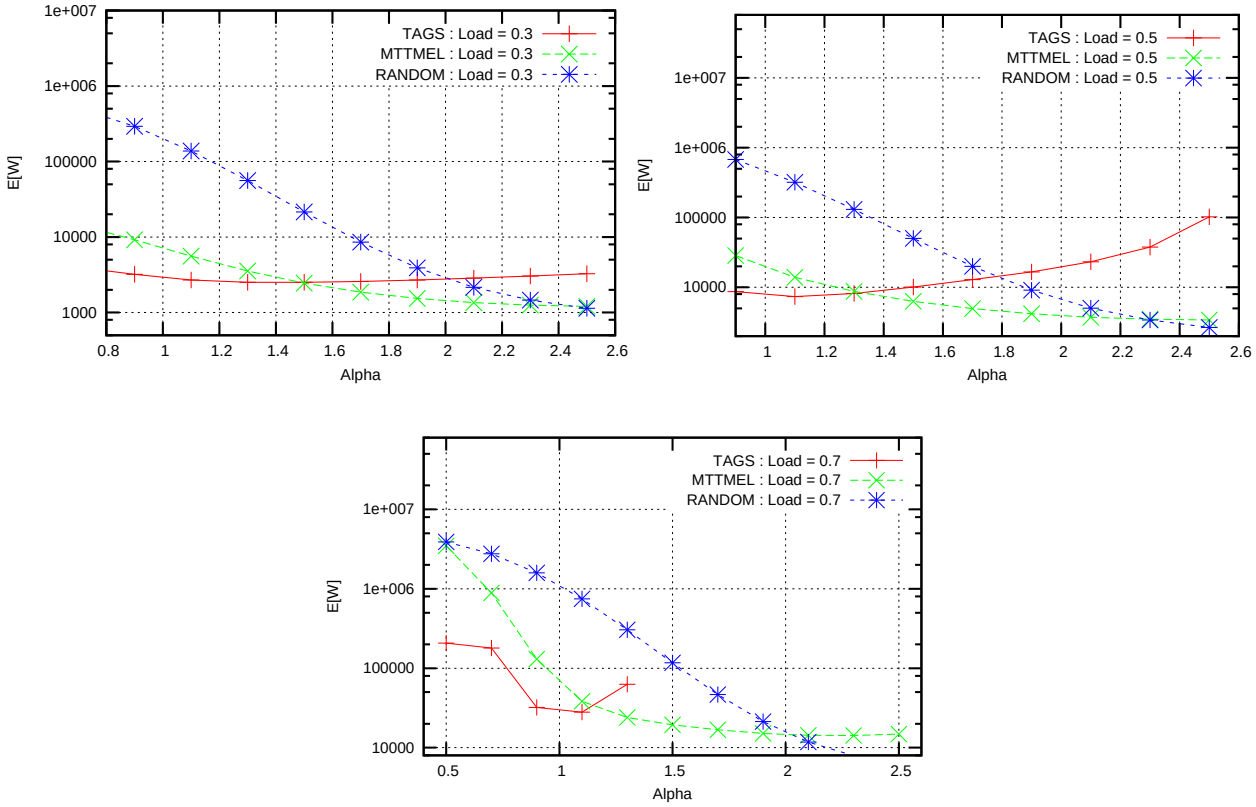


Figure 5.3: Expected waiting time for MTTMEL in a 3 Host system

range of α values considered. We also note that under a system load of 0.5, TAGS's performance deteriorates considerably if α exceeds a certain value. This means that in a 3 Host system, TAGS can only perform well if α is low. Finally, note that as α approaches 0.5, TAGS begins to outperform MTTMEL. However, the factor of improvement is not highly significant.

Let us now consider the performance of MTTMEL in a 6 Host system with three tiers, where Tier 1 has three hosts, Tier 2 has two hosts and Tier 3 has one host. The average arrival rates into Tiers, λ_1, λ_2 and λ_3 are given by

$$\begin{aligned} \lambda_1 &= \frac{\lambda}{3}, \\ \lambda_2 &= \frac{\lambda}{2}(p_2 + p_3), \\ \lambda_3 &= \lambda p_3, \end{aligned} \tag{5.9}$$

where λ denotes the average arrival rate into the system (given by Equation 5.7) and p_i is given by Equation 5.1. Figure 5.4 illustrates the performance of the policies considered in a 6 Host system. First we note that MTTMEL performs consistently well in steady state in both 3 and 6 Host systems

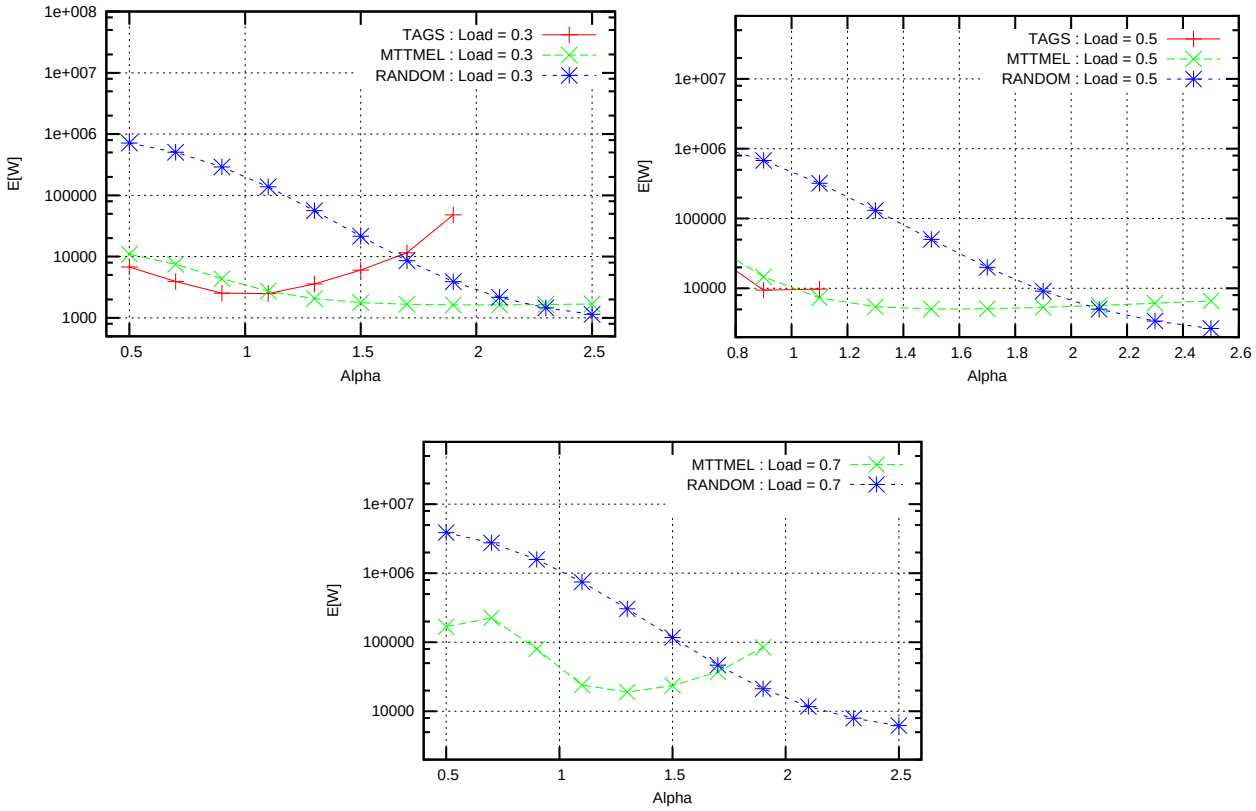


Figure 5.4: Expected waiting time for MTTMEL in a 6 Host system

for almost all the scenarios considered. Second we note that the factor of improvement in TAGS over MTTMEL is not significant under low α values. Third we note that under moderate (0.5) system loads, TAGS can operate in steady state only if α is low. When the system load is high (0.7), TAGS cannot operate in steady state at all. This means that for a wide range of α values we cannot find cut-offs so that load on each host is less than one.

5.1.4 Analysis of excess load under MTTMEL

One of the important characteristics of MTTMEL is that its performance does not degrade under low and moderate task size variabilities, and high system loads, because MTTMEL does not generate

large amounts of excess load. This section investigates the excess load generated under the TAGS and MTTMEL. Let us first derive an expression for the excess load for MTTMEL.

Let Z_i be the true load on a host in Tier i

$$Z_i = \lambda_i E[X_{iv}]. \tag{5.10}$$

Let L_{sum} be the true sum of loads in the system

$$L_{sum} = \sum_{i=1}^{T} Z_i T_i, \tag{5.11}$$

where T_i denotes the number of hosts in Tier i. The excess load in the system is the difference between the true sum of loads and expected sum of loads, i.e.

$$Excess = L_{sum} - \lambda E[X]. \tag{5.12}$$

Figures 5.5 and 5.6 plot the excess load of TAGS and MTTMEL in 3 and 6 Host systems. We note

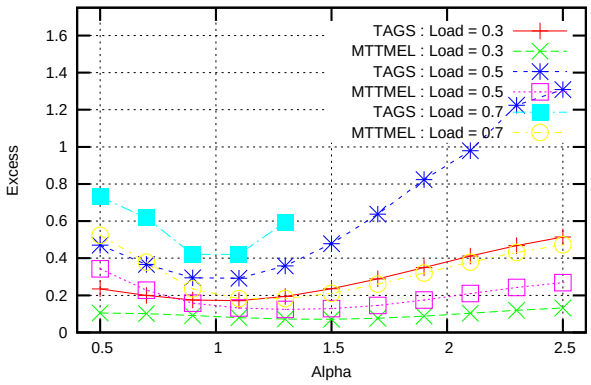


Figure 5.5: Excess load under MTTMEL in a 3 Host system

that the excess load under MTTMEL is significantly less than that of TAGS. For example, in a 3 Host system when $\alpha = 0.5$ and system load = 0.5, the ratio between TAGS excess and MTTMEL excess is equal to 1.3. Under the same conditions, when α equals 2.1, this ratio is equal to 4.6.

Under TAGS, large tasks are killed and restarted from scratch many times, whereas in a MTTMEL large tasks are killed and restarted from scratch only a few times. For example, a six hosts TAGS system may kill and restart large tasks up to 5 times, while six hosts MTTMEL system may kill

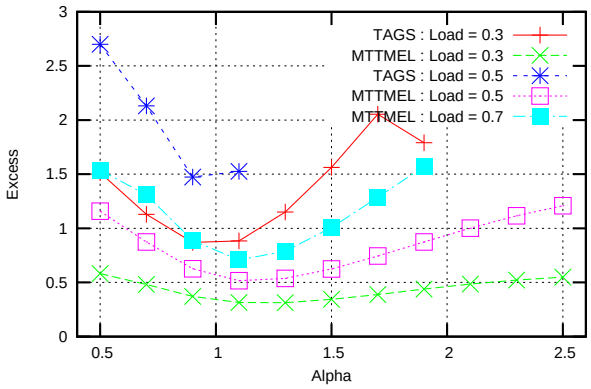


Figure 5.6: Excess load under MTTMEL in a 6 Host system

and restart large tasks only up to maximum of 2 times. MTTMEL, therefore, generates less excess load compared to that of TAGS. Under certain conditions (e.g. large-size server farms, high system loads, etc.), it is possible for MTTMEL to perform poorly. In this case, it is possible to improve the performance by changing the number of tiers and number of servers in tiers (based on the properties of the traffic).

5.2 Multi-cluster Task Assignment based on Preemptive Migration (MCTPM)

This section presents the details of Multi-cluster Task Assignment based on Preemptive Migration (MCTPM) policy, an efficient task assignment policy for assigning tasks in multiple server farms. In the previous section we noted that MTTMEL has certain desirable properties over TAGS in stand-alone server farms. However, MTTMEL restarted tasks from scratch resulting in some excess load on the system. In this section we extend MTTMEL proposed for stand-alone server farms in the previous section and propose a new task assignment policy for assigning tasks in a multiple server farms.

5.2.1 Overview of MCTPM

MCTPM is designed for assigning tasks in multiple server farm environments (i.e. a set of server farms). Figure 5.7 illustrates the proposed policy for the simple case of two server farms, where each farm consists of five hosts and two tiers.[5] The notation presented in Table 5.1 is used to describe

[5]Note that the performance model we present in Section 5.2.2 is a general model, which is valid for an arbitrary number of server farms with an arbitrary number of tiers.

MCTPM.

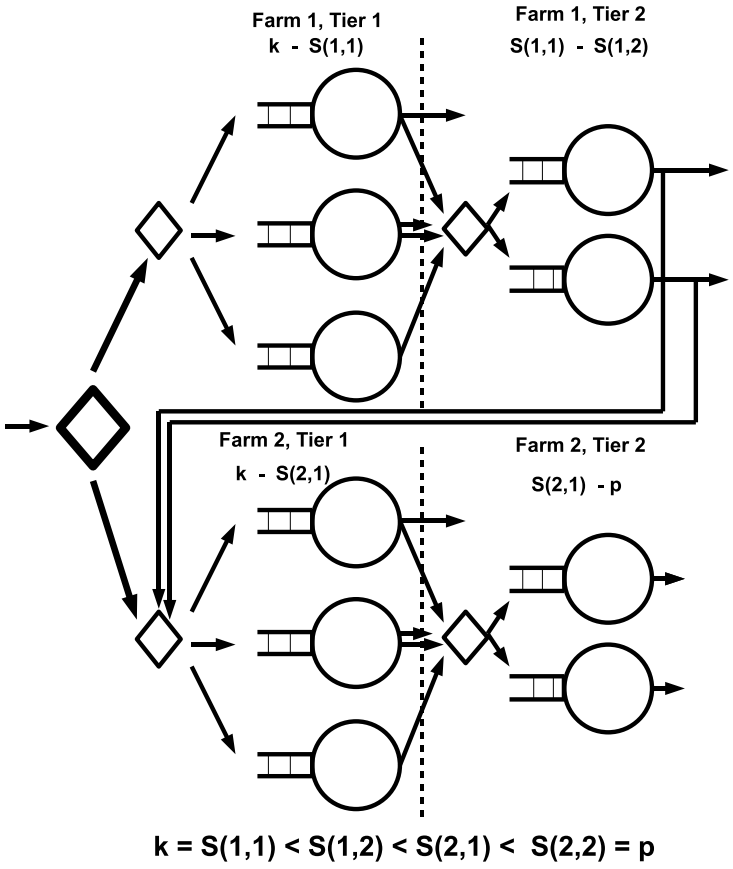


Figure 5.7: MCTPM architecture

As can be seen from Figure 5.7, MCTPM consists of a global task dispatcher, where the incoming tasks first arrive. The global dispatcher assigns these tasks among a set of server farms with probability q_i, where q_i is the fraction of tasks assigned to Farm i. We note that MCTPM consists of multiple tiers of hosts in farms. The hosts in a particular tier process tasks only up to a predefined time limit assigned to the tier. The relationship among these time limits is given by

$$\begin{aligned} &s_{(1,1)} < s_{(1,2)} < \ldots < s_{(1,N_1-1)} < s_{(1,N_1)} < s_{(2,1)} < s_{(2,2)} \\ &\ldots < s_{(2,N_2-1)} < s_{(2,N_2)} < \ldots < s_{(N,N_n-1)} < s_{(N,N_n)} = p. \end{aligned} \tag{5.13}$$

The core functionality of MCTPM policy can be summarised as follows.

- Each new task arrives at the global dispatcher.

N_i	Number of tiers in Farm i
n	Number of farms
$s_{(i,j)}$	Designated time limit assigned Farm i's Tier j
$s_{(i,j-1)} - s_{(i,j)}$	Task size range for Farm i's Tier j
$f(x)$	Service time distribution of tasks (i.e. $B(k,p,\alpha)$)
k	Lower bound of the task size distribution
p	Upper bound of the task size distribution
λ	Average arrival rate of tasks at the global dispatcher
α	Heavy-tail parameter
$E[X^l_{DF:i,j}]$	l^{th} moment of the remaining processing time distribution of a task dispatched to Farm i from the global dispatcher (D) whose final (F) destination is a host in Farm i's Tier j
$E[X^l_{PF:i,j}]$	l^{th} moment of the remaining processing time distribution of a task that arrives at Farm i from the predecessor (P) farm whose final (F) destination is a host in Farm i's Tier j
$E[X^l_{S:i,j}]$	l^{th} moment of the remaining processing time distribution of a task that spends time (S) at a host in Farm i's Tier j
$E[X_{D\bar{F}:i,j}]$	Expected size of a task that arrives directly from the global dispatcher (D) that does not run to completion ($\bar{F}$ for 'not final') at a host in Farm i's Tier j
$E[X_{P\bar{F}:i,j}]$	Expected size of a task that arrives from a host in the predecessor (P) farm that does not run to completion ($\bar{F}$ for 'not final') at a host in Farm i's Tier j
$N_{(i,j)}$	denotes the number of hosts in Farm i's Tier j
$E[W_{(i,j)}]$	Expected waiting time of a task that spends time at a host in Farm i's Tier j
$E[W]$	Expected waiting time in the system (i.e. overall expected waiting time)
γ_d	Fixed migration cost (resumption cost) incurs on a destination host when migrating a task between two consecutive tiers
γ_s	Fixed migration cost (transfer cost) incurs on a source host when migrating a task between two consecutive tiers
β_d	Proportional migration cost (resumption cost) incurs on a destination host when migrating a task between two consecutive tiers
β_s	Proportional migration cost (transfer cost) incurs on a source host when migrating a task between two consecutive tiers

Table 5.1: Notation: MCTPM

- The global dispatcher assigns tasks to Farm 1 with probability q_1, Farm 2 with probability q_2 and so on.
  - Farm 1's Dispatcher 1 assigns each task to Farm 1's Tier 1 with an equal probability. Each task is serviced in a FCFS manner up to a predefined time limit assigned to Farm 1's Tier 1.
  - If the service time of a task is less than or equal to the predefined time limit assigned to Farm 1's Tier 1, the task departs the system and the results are sent back to the client.

  - If the service time of the task is greater than the predefined time limit assigned to Farm 1's Tier 1, the task is sent to Farm 1's Dispatcher 2, which assigns the task to a host in Farm 1's Tier 2 with an equal probability.
  - The task is serviced in a FCFS manner up to the predefined time limit of Farm 1's Tier 2. When the task receives more service in Farm 1's Tier 2, its execution is resumed (i.e. preemptive migration) and so on.
  - If the size of a task is less than or equal to the predefined time limit assigned to Farm 1's final tier (i.e. $s_{(1,N_1)}$), the task departs the system from Farm 1's final tier and the results are sent back to the client.
  - If the service time of the task is greater than the predefined time limit assigned to Farm 1's final tier, the task is migrated to Farm 2.
  - The task arrives at Farm 2's Dispatcher 1 and processed in a similar manner. When the task receives more service at Farm 2, its execution is resumed (i.e. preemptive migration).

- The tasks that are directly assigned to Farm 2 by the global dispatcher are also processed in a similar manner and so on.

- As indicated in Figure 5.7, except for Farm 1, each farm receives tasks from its predecessor farm and from the global dispatcher. The tasks that arrive from predecessor farms have already received some service, whereas the task that arrive from directly from the global dispatcher are new tasks, which have so far received no service.

- The size ranges for tiers and the fractions of tasks dispatched to farms are computed to optimise a certain performance criteria such as the expected waiting time and expected slowdown. In this chapter as pointed out earlier, we use the expected waiting time as the key performance metric.[6]

An important property of MCTPM is that it supports preemptive task migration between hosts in consecutive tiers. In the previous chapter we considered preemptive migration in time sharing server farms. The main difference between the preemptive migration presented in this chapter and the previous chapter is that in the previous chapter we assumed that migration cost is negligible, whereas in this chapter we assume that migration cost is not zero (which is typically the case for tasks processed in batch computing environments). It is often the case that when a task is migrated from one host to another, two types of costs incur on the system, namely, the fixed migration cost and the

[6] In the case of expected slowdown we expect to get similar or better results.

proportional migration cost [Milojicic et al., 2000; Broberg et al., 2006]. The fixed cost is associated with the transfer of state information, while the proportional migration cost is associated with the transfer of memory [Milojicic et al., 2000]. The fixed cost is a fixed processing time computed based upon the mean of the service time distribution. The proportional migration cost, on the other hand, is a fraction, which is used to compute a processing time proportional to the task size. When a task is migrated from a particular host in one server farm to a particular host in the consecutive server farm, it is possible that there is some delay between the time that the time task arrives at the destination host and the time that the task departs the source host. In this chapter we assume that this delay is at its minimal levels and hence can be equated to zero.[7]

5.2.2 Performance model for MCTPM

This section presents an analytical performance model for the proposed policy by applying queueing theory. This performance model is based on the notation given in Table 5.1. The final result of this performance model is the derivation of $E[W]$, the overall expected waiting time of a task in the system, which is a function of expected waiting time of a task that spends time at a host in Farm i's Tier j (i.e. $E[W_{(i,j)}]$).[8] $E[W_{(i,j)}]$ is given by the Pollaczek-Khinchin formula (see Equation 5.38) and it is a function of $E[X^l_{S:i,j}]$, l^{th} moment of the remaining processing time distribution of a task that spends time at a host in Farm i's Tier j. $E[X^l_{S:i,j}]$ is based on $E[X^l_{DF:i,j}]$, $E[X^l_{PF:i,j}]$, $E[X_{D\bar{F}:i,j}]$ and $E[X_{P\bar{F}:i,j}]$ (refer to Table 5.1). The first step is to derive these four moments.

Let $p_{(i,j)}$ be the probability that the service time of a task is between $s_{(i,j-1)}$ and $s_{(i,j)}$, then

$$p_{(i,j)} = \begin{cases} \int_{s_{(i,j-1)}}^{s_{(i,j)}} f(x)dx, & j > 1, \\ \int_{s_{(i-1,N_{i-1})}}^{s_{(i,j)}} f(x)dx, & j = 1. \end{cases} \tag{5.14}$$

We define p_i as follows:

$$p_i = \sum_{k=1}^{N_i} p_{(i,k)}. \tag{5.15}$$

Let $E[X^l_{DF:(i,j)}]$ be the l^{th} moment of the remaining processing time distribution of a task dispatched to Farm i from the global dispatcher (D) whose final (F) destination is a host in Farm i's Tier j, then,

$$E[X^l_{DF:i,j}] = \begin{cases} \frac{\int_k^{s_{(i,j)}} x^l f(x)dx}{(\sum_{k=1}^{i-1} p_k + p_{(i,j)})}, & i \geq 1, j = 1, \\ \frac{\int_{s_{(i,j-1)}}^{s_{(i,j)}} (x - s_{(i,j-1)} + \gamma_d + \beta_d x)^l f(x)dx}{p_{(i,j)}}, & i \geq 1, j > 1. \end{cases} \tag{5.16}$$

[7] In order for this condition to be true it may be necessary for the set of server farms to be located in the close proximity to each other. In the case where there are significant delays, additional delay parameters can be included into the performance equations to cater for such delays. In this chapter we do not consider this.

[8] Note that the system consists of n number of farms.

Let $E[X^l_{PF:i,j}]$ be the l^{th} moment of the remaining processing time distribution of a task that arrives at Farm i from the predecessor (P) farm whose final (F) destination is a host in Farm i's tier j. Note that Farm 1 has no predecessor hosts. $E[X^l_{PF:i,j}]$ is given by

$$E[X^l_{PF:i,j}] = \begin{cases} \frac{\int_{s_{(i-1,N_{i-1})}}^{s_{(i,1)}} (x - s_{(i-1,N_{i-1})} + \gamma_d + \beta_d x)^l f(x) dx}{p_{(i,1)}}, & i > 1,\ j = 1, \\ \frac{\int_{s_{(i,j-1)}}^{s_{(i,j)}} (x - s_{(i,j-1)} + \gamma_d + \beta_d x)^l f(x) dx}{p_{(i,j)}}, & i > 1,\ j > 1. \end{cases} \tag{5.17}$$

We note that for $i > 1$ and $j > 1$,

$$E[X^l_{DF:i,j}] = E[X^l_{PF:i,j}]. \tag{5.18}$$

But for $i > 1$ and $j = 1$,

$$E[X^l_{DF:i,j}] \neq E[X^l_{PF:i,j}]. \tag{5.19}$$

Let $E[X_{D\bar{F}:i,j}]$ be the expected size of a task that arrives directly from the global dispatcher (D) that does not run to completion at a host in Farm i's Tier j and let $E[X_{P\bar{F}:i,j}]$ be the expected size of a task that arrives from a host in the predecessor (P) farm that does not run to completion at a host in Farm i's Tier j. We obtain

$$E[X_{D\bar{F}:i,j}] = \begin{cases} (1 - (\sum_{k=1}^{i-1} p_k + p_{(i,j)})) \int_{s_{(i,j)}}^{p} x f(x) dx, & i \geq 1, j = 1, \\ \frac{q_i(\sum_{k=i+1}^{n} p_k + \sum_{k=j}^{N_i} p_{(i,k)}) - q_i p_{(i,j)}}{q_i(\sum_{k=i+1}^{n} p_k + \sum_{k=j}^{N_i} p_{(i,k)})} \int_{s_{(i,j)}}^{p} x f(x) dx, & i \geq 1, j > 1, \end{cases} \tag{5.20}$$

and

$$E[X_{P\bar{F}:i,j}] = \frac{(\sum_{k=1}^{i-1} q_k)(\sum_{k=i+1}^{n} p_n + \sum_{k=j}^{N_i} p_{(i,j)}) - (\sum_{k=1}^{i-1} q_k) p_{(i,j)}}{(\sum_{k=1}^{i-1} q_k)(\sum_{k=i+1}^{n} p_k + \sum_{k=j}^{N_i} p_{(i,j)})} \int_{s_{(i,j)}}^{p} x f(x) dx, \quad i \geq 1, j \geq 1. \tag{5.21}$$

We can now derive $E[W_{S:i,j}]$, l^{th} moment of the processing time distribution of a task that spends (S) time at a host in Farm i's Tier j. $E[W_{S:i,j}]$ is based on the four probabilities shown in Table 5.2.

$p_{DF:i,j}$	The probability that the task arrives directly from the global dispatcher and the final destination is a host in Farm i's Tier j
$p_{D\bar{F}:i,j}$	The probability that the task arrives directly from the global dispatcher and the final destination is not a host in Farm i's Tier j
$p_{PF:i,j}$	The probability that the task arrives from a predecessor farm and the final destination is a host in Farm i's Tier j
$p_{P\bar{F}:i,j}$	The probability that the task arrives from a predecessor farm and the final destination is not a host in Farm i's Tier j

Table 5.2: Probabilities for MCTPM

When $i = 1$ (i.e. Farm 1) and $j = 1$ (i.e. Tier 1),

$$E[X^l_{S:i,j}] = p_{DF:i,j} E[X^l_{DF:i,j}] + p_{D\bar{F}:i,j} (s_{(i,j)} + \beta_s E[X_{D\bar{F}:i,j}] + \gamma_s)^l, \tag{5.22}$$

where

$$p_{DF:i,j} = p_{(i,j)}, \tag{5.23}$$

$$p_{D\bar{F}:i,j} = 1 - p_{(i,j)}. \tag{5.24}$$

When $i > 1$ and $j = 1$,

$$\begin{aligned} E[X^l_{S:i,j}] =& p_{DF:i,j}E[X^l_{DF:i,j}] + p_{D\bar{F}:i,j}(\beta_s E[X_{D\bar{F}:i,j}] + \gamma_s + s_{(i,j)})^l + \\ & p_{PF:i,j}E[X^l_{PF:i,j}] + p_{P\bar{F}:i,j}\left(s_{(i,j)} - s_{(i-1,N_{i-1})} + (\beta_d + \beta_s)E[X_{P\bar{F}:i,j}] + \gamma_s + \gamma_d\right)^l, \end{aligned} \tag{5.25}$$

where

$$p_{DF:i,j} = \frac{q_i(\sum_{k=1}^{i-1} p_k + p_{(i,j)})}{q_i + (\sum_{k=1}^{i-1} q_k)(\sum_{k=i}^{n} p_k)}, \tag{5.26}$$

$$p_{D\bar{F}:i,j} = \frac{q_i - q_i(\sum_{k=1}^{i-1} p_k + p_{(i,j)})}{q_i + (\sum_{k=1}^{i-1} q_k)(\sum_{k=i}^{n} p_k)}. \tag{5.27}$$

Note that all tasks that are processed in Farm i's Tier 1, that arrive from the global dispatcher are new tasks that have so far received no service. $p_{PF:i,j}$ and $p_{P\bar{F}:i,j}$ are given by

$$p_{PF:i,j} = \frac{(\sum_{k=1}^{i-1} q_k)p_{(i,j)}}{q_i + (\sum_{k=1}^{i-1} q_k)(\sum_{k=i}^{n} p_k)}, \tag{5.28}$$

$$p_{P\bar{F}:i,j} = \frac{(\sum_{k=1}^{i-1} q_k)(\sum_{k=i}^{n} p_k) - (\sum_{k=1}^{i-1} q_k)p_{(i,j)}}{q_i + (\sum_{k=1}^{i-1} q_k)(\sum_{k=i}^{n} p_k)}. \tag{5.29}$$

The denominator of four equations above represents probability that a task is processed at Farm i's Tier 1.

When $i > 1$ and $j > 1$,

$$\begin{aligned} E[X^l_{S:i,j}] =& p_{DF:i,j}E[X^l_{DF:i,j}] + p_{D\bar{F}:i,j}\left((\beta_d + \beta_s)E[X_{D\bar{F}:i,j}] + \gamma_s + \gamma_d + s_{(i,j)} - s_{(i,j-1)}\right)^l + \\ & p_{PF:i,j}E[X^l_{PF:i,j}] + p_{P\bar{F}:i,j}\left((\beta_d + \beta_s)E[X_{P\bar{F}:i,j}] + \gamma_s + \gamma_d + s_{(i,j)} - s_{(i,j-1)}\right)^l, \end{aligned} \tag{5.30}$$

where

$$p_{DF:i,j} = \frac{q_i p_{(i,j)}}{(q_i + \sum_{k=1}^{i-1} q_k)(\sum_{k=i+1}^{n} p_k + \sum_{k=j}^{N_i} p_{(i,k)})}, \tag{5.31}$$

$$p_{D\bar{F}:i,j} = \frac{q_i(\sum_{k=i+1}^{n} p_k + \sum_{k=j}^{N_i} p_{(i,k)}) - q_i p_{(i,j)}}{(q_i + \sum_{k=1}^{i-1} q_k)(\sum_{k=i+1}^{n} p_k + \sum_{k=j}^{N_i} p_{(i,k)})}, \tag{5.32}$$

$$p_{PF:i,j} = \frac{(\sum_{k=1}^{i-1} q_k)p_{(i,j)}}{(q_i + \sum_{k=1}^{i-1} q_k)(\sum_{k=i+1}^{n} p_k + \sum_{k=j}^{N_i} p_{(i,k)})}, \tag{5.33}$$

$$p_{P\bar{F}:i,j} = \frac{(\sum_{k=1}^{i-1} q_k)(\sum_{k=i+1}^{n} p_k + \sum_{k=j}^{N_i} p_{(i,k)}) - (\sum_{k=1}^{i-1} q_k)p_{(i,j)}}{(q_i + \sum_{k=1}^{i-1} q_k)(\sum_{k=i+1}^{n} p_k + \sum_{k=j}^{N_i} p_{(i,k)})}. \tag{5.34}$$

When $i = 1$ and $j > 1$,

$$E[X^l_{S:i,j}] = p_{DF:i,j}E[X^l_{DF:i,j}] + p_{D\bar{F}:i,j}\left((\beta_d+\beta_s)E[X_{D\bar{F}:i,j}] + \gamma_s + \gamma_d + s_{(i,j)} - s_{(i,j-1)}\right)^l, \tag{5.35}$$

where

$$p_{DF:i,j} = \frac{q_i p_{(i,j)}}{q_i(\sum_{k=i+1}^{n} p_k + \sum_{k=j}^{N_i} p_{(i,k)})}, \tag{5.36}$$

$$p_{D\bar{F}:i,j} = \frac{q_i(\sum_{k=i+1}^{n} p_k + \sum_{k=j}^{N_i} p_{(i,k)}) - q_i p_{(i,j)}}{q_i(\sum_{k=i+1}^{n} p_k + \sum_{k=j}^{N_i} p_{(i,k)})}. \tag{5.37}$$

Note that $p_{DF:i,j} + p_{D\bar{F}:i,j} + p_{PF:i,j} + p_{P\bar{F}:i,j} = 1$ for all i, j.

Let $E[W_{(i,j)}]$ be the expected waiting time of a task that spends time at a host in Farm i's tier j. $E[W_{(i,j)}]$ is obtained by applying the Pollaczek-Khinchin formula [Kleinrock, 1975],

$$E[W_{(i,j)}] = \frac{\lambda_{i,j}E[X^2_{S:i,j}]}{2(1-\lambda_{i,j}E[X_{S:i,j}])}, \tag{5.38}$$

where $\lambda_{i,j} = \lambda_{D:i,j} + \lambda_{P:i,j}$ and $\lambda_{D:i,j}$ denotes the average arrival rate of tasks into Farm i's Tier j from the global dispatcher, while $\lambda_{P:i,j}$ denotes the average arrival rate of tasks into Farm i's Tier j from Farm $i-1$. We obtain

$$\lambda_{D:i,j} = \begin{cases} \frac{\lambda q_i}{N_{(i,j)}}, & i \geq 1, j = 1, \\ \frac{\lambda q_i(1-\sum_{k=1}^{i} p_k + \sum_{k=j}^{N_i} p_{(i,k)}))}{N_{(i,j)}}, & i \geq 1, j > 1, \end{cases} \tag{5.39}$$

and

$$\lambda_{P:i,j} = \frac{\lambda \sum_{k=1}^{i-1} q_k(1-\sum_{k=1}^{i} p_k + \sum_{k=j}^{N_i} p_{(i,k)})}{N_{(i,j)}}, \qquad i > 1, j \geq 1. \tag{5.40}$$

Let $E[W_D]$ be the overall expected waiting time of a task that arrives directly from the dispatcher. $E[W_D]$ is given by

$$\begin{aligned} E[W_D] =& q_1(\sum_{m=1}^{N_1} p(1,m) \sum_{k=1}^{m} E[W_{(1,k)}]) \\ &+ q_2((p_1 + p_{(2,1)})E[W_{(2,1)}] + \sum_{m=2}^{N_2} p_{(2,m)} \sum_{k=1}^{m} E[W_{(2,k)}]) \\ &+ ... \\ &+ q_n((\sum_{k=1}^{N-1} p_k + p_{(N,1)})E[W_{(N,1)}] + \sum_{m=2}^{N_n} p_{(N,m)} \sum_{k=1}^{m} E[W_{(N,k)}]). \end{aligned} \tag{5.41}$$

Let $E[W_{\bar{D}}]$ be the overall expected waiting time of a task that does not arrive directly from the dis-

patcher. $E[W_{\bar{D}}]$ is given by

$$\begin{aligned} E[W_{\bar{D}}] =& q_1(\sum_{m=1}^{N_2} p_{(2,m)}(\sum_{k=1}^{m} E[W_{(2,k)}] + \sum_{k=1}^{N_1} E[W_{(1,k)}])) \\ &+ (q_1+q_2)(\sum_{m=1}^{N_3} p_{(3,m)}(\sum_{k=1}^{m} E[W_{(3,k)}] + \sum_{T=1}^{2}\sum_{k=1}^{N_T} E[W_{(T,k)}])) \\ &+ ... \\ &+ (\sum_{d=1}^{N-1} q_d)(\sum_{m=1}^{N_n} p_{(N,m)}(\sum_{k=1}^{m} E[W_{(N,k)}] + \sum_{T=1}^{N-1}\sum_{k=1}^{N_T} E[W_{(T,k)}])). \end{aligned} \tag{5.42}$$

Let $E[W]$ be the overall expected waiting time of a task in the system. We obtain

$$E[W] = E[W_D] + E[W_{\bar{D}}]. \tag{5.43}$$

The cut-offs (i.e. $s_{(i,j)}$ values), and the fraction of tasks assigned to server farms (i.e. q_i values) are computed to optimise the $E[W]$ for a given system load and α. The system load, ρ, is given by

$$\rho = \frac{\lambda E[X]}{\sum_{i=1}^{n}\sum_{j=1}^{N_i} N_{(i,j)}}, \qquad \rho < 1, \tag{5.44}$$

where λ, $E[X]$ and $N_{(i,j)}$ are the average arrival rate at the global dispatcher, mean of the service time distribution and number of hosts in Farm i's Tier j respectively. The performance evaluation is carried out under three important system loads: low (0.3), moderate (0.5) and high (0.7).

5.2.3 Performance evaluation: MCTPM

This section evaluates the performance of the proposed policy by comparing its performance with three important task assignment policies, namely, MC-Random, MC-TAGSPM (Multi-cluster Task assignment by Guessing Size based on Preemptive Migration) and MC-MTTPM (Multi-cluster-multi-tier Task assignment based on Preemptive Migration). These policies have different performance characteristics under difference workload conditions. Figure 5.8 illustrates the host architectures of these policies for a simple case, where the number of server farms = 2 and the number of servers in a farm = 3. A brief description of each of these policies is as follows.

- MC-Random: MC-Random is based on the simplest Random task assignment policy, which we discussed in Section 2.3. Under MC-Random, new tasks arrive at a global dispatcher. The global dispatcher directs these tasks to the central dispatchers of stand-alone server farms with an equal probability. The local dispatchers of server farms distribute these new tasks among their hosts with an equal probability. These tasks are processed at back-end hosts in a FCFS

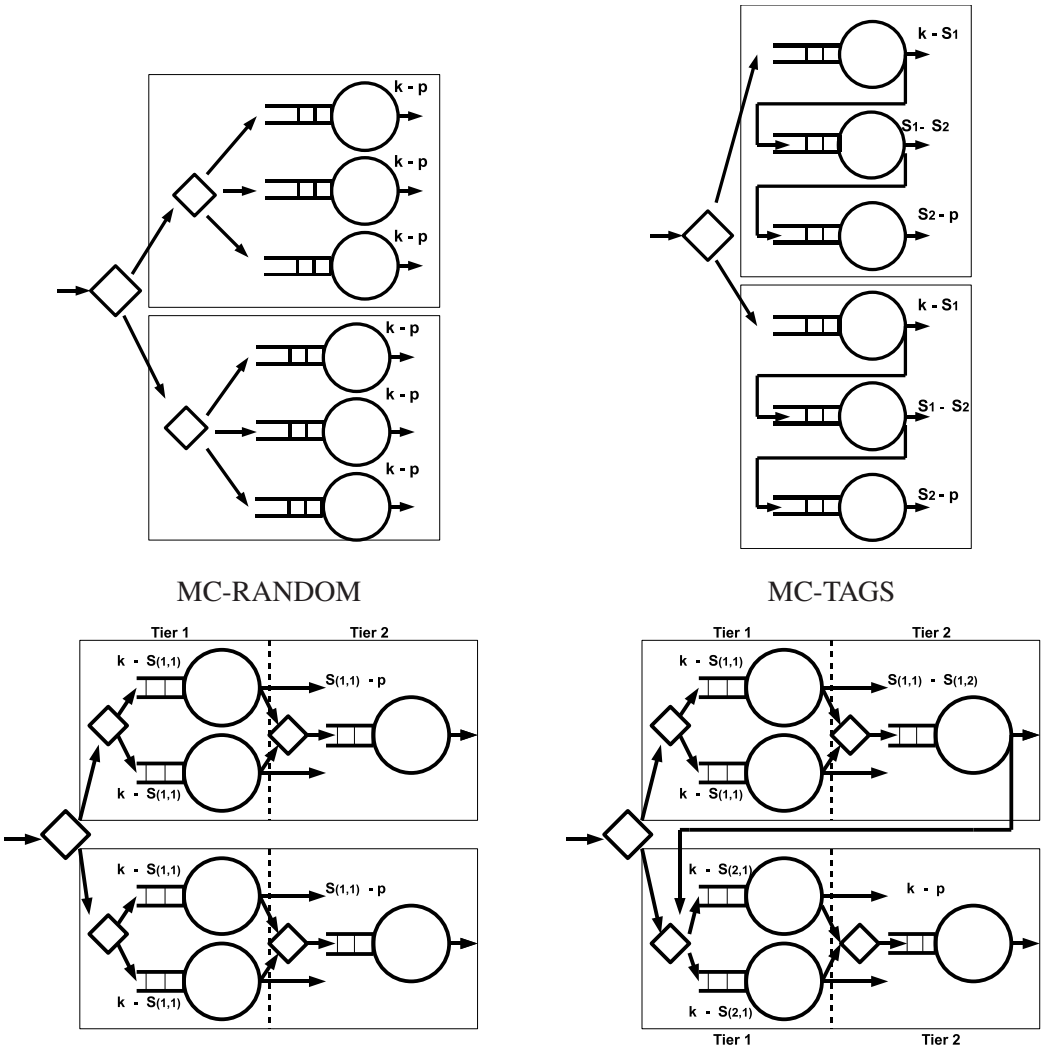


Figure 5.8: Task assignment policies for multiple server farms

manner until completion. Since the average arrivals rates into hosts and the processing time distributions of tasks seen by hosts are same for all hosts in all server farms, the expected waiting time in the system is equal to the expected waiting time of a task at any given host. This can be obtained using Pollaczek-Khinchin formula [Kleinrock, 1975] (see Equation 6.17).

- MC-TAGSPM: This is based on the popular TAGS-PM [Harchol-Balter, 2002], which we discussed in Section 2.4.2. In the case of MC-TAGSPM the global dispatcher assigns the incoming

tasks to Host 1 of each server farm with an equal probability. Tasks are processed according to TAGS-PM at farms. The cut-offs for MC-TAGSPM is computed to optimise the expected waiting time.

- MC-MTTPM: To be compatible with MCTPM, we introduce this new task assignment policy, which is also based on a multi-tier host architecture. There are three main differences between MC-MTTPM and MCTPM: 1) MC-MTTPM does not support task migration between server farms. However, it does support preemptive migration within the server farm (i.e. between hosts in different tiers), 2) MC-MTTPM always assigns tasks into server farms with an equal probability and it cannot control the traffic flow into server farms and 3) The processing time limits for MC-MTTPM are computed by considering the individual host architectures of server farms, whereas the processing time limits of MCTPM are computed by considering the host architectures of all the server farms in the system. The time limits for tiers are computed to optimise the expected waiting time.

- MCTPM: MCTPM is the proposed policy details of which can be found in Section 5.2.

The expected waiting time under above policies are evaluated under three distinct proportional cost criteria: $\beta = \beta_s = \beta_d = 0.25$ (25% of task size), $\beta = \beta_s = \beta_d = 0.50$ (50% of task size) and $\beta = \beta_s = \beta_d = 0.75$ (75% of task size) and three distinct fixed cost criteria: $\gamma = \gamma_s = \gamma_d = 750$ (25% of mean), $\gamma = \gamma_s = \gamma_d = 1250$ (50% of mean) and $\gamma = \gamma_s = \gamma_d = 2250$ (75% of mean). Note that the fixed cost represents a fixed processing time computed based upon the mean of the service time distribution, while the proportional cost is a fraction, which is used to compute a processing time proportional to the task size.

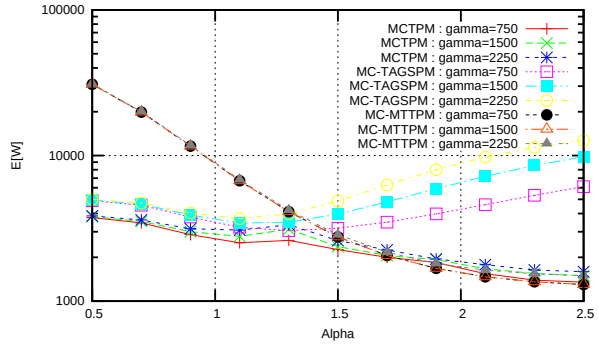
5.2.4 Expected waiting time in small-sized server farms

This section investigates the expected waiting time for policies in two server farms when the number of hosts in each farm is equal to three. The primary objective is to compare the expected waiting time for MCTPM with other policies in small sized-server farms under different workload scenarios.

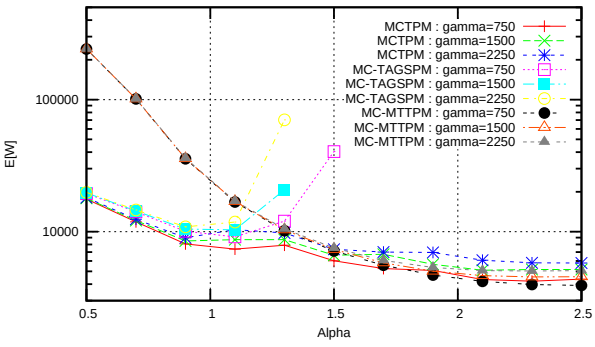
It is important to point out that we exclude the results for MC-Random from all the plots for the purpose of improving the clarity plots. MC-Random performs very poorly compared to other three policies over a wide range of scenarios. The specific cases where MC-Random outperforms other policies will be discussed.

5.2.4.1 Low and moderate system loads

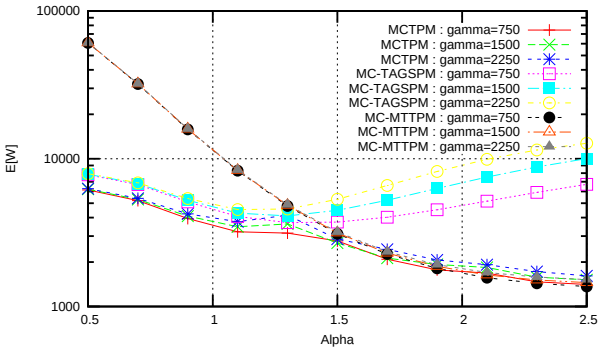
As shown in Figure 5.9, the behaviour of expected waiting time of policies under low and moderate systems loads has similar characteristics. Let us first compare the expected waiting time under



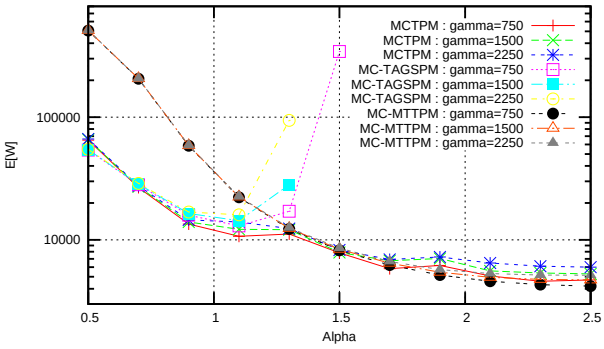
(a) System load = 0.3, $\beta = 0.25$



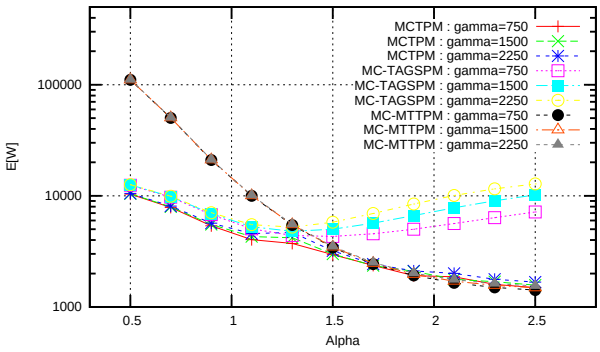
(b) System load = 0.5, $\beta = 0.25$



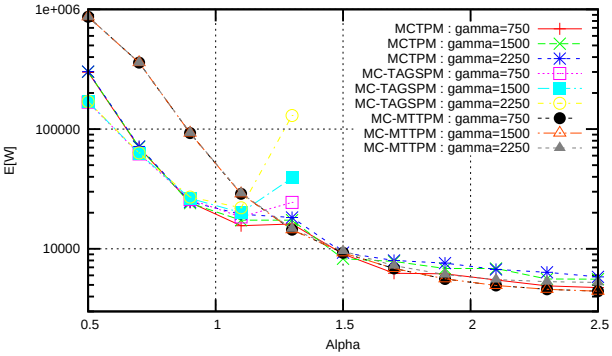
(a) System load = 0.3, $\beta = 0.50$



(b) System load = 0.5 , $\beta = 0.50$



(a) System load = 0.3, $\beta = 0.75$



(b) System load = 0.5 , $\beta = 0.75$

Figure 5.9: Expected waiting time for MCTPM in small-sized server farms: system load = 0.3 *and* 0.5

MCTPM and MC-MTTPM. We note that under low α values, MCTPM performs significantly better

than MC-MTTPM. For example, under a system load of 0.3, when α = 0.5, β = 0.75 and γ = 750, MCTPM outperforms MC-MTTPM by a factor of 10.5. We note that when α is in the range 1.5 - 2.5, the expected waiting time under two policies are very similar. In fact, as α approaches 2.5, MC-MTTPM starts to outperform MCTPM. The factor of improvement, however, is not significant and does not exceed 1.10 under any given migration scenario.

Let us now compare the performance of MC-TAGSPM and MCTPM. MCTPM performs exceptionally well compared to MC-TAGSPM under high α values ($\alpha > 1.5$).[9] For example, under a system load of 0.5, when $\alpha = 2.5$, β = 0.75 and γ = 750, MCTPM outperforms MC-TAGSPM by a factor of 5. Under low α values ($\alpha < 1.5$), the factor of improvement in MCTPM over MC-TAGSPM is relatively small compared to that of high α values. For example, under a system load of 0.3, when $\alpha = 0.5$, β = 0.25 and γ = 750, MCTPM outperforms MC-TAGSPM by a factor of 1.3. Under very specific scenarios (e.g. α = 0.5, $\beta = 0.5$), MC-TAGSPM performs slightly better compared to MCTPM.

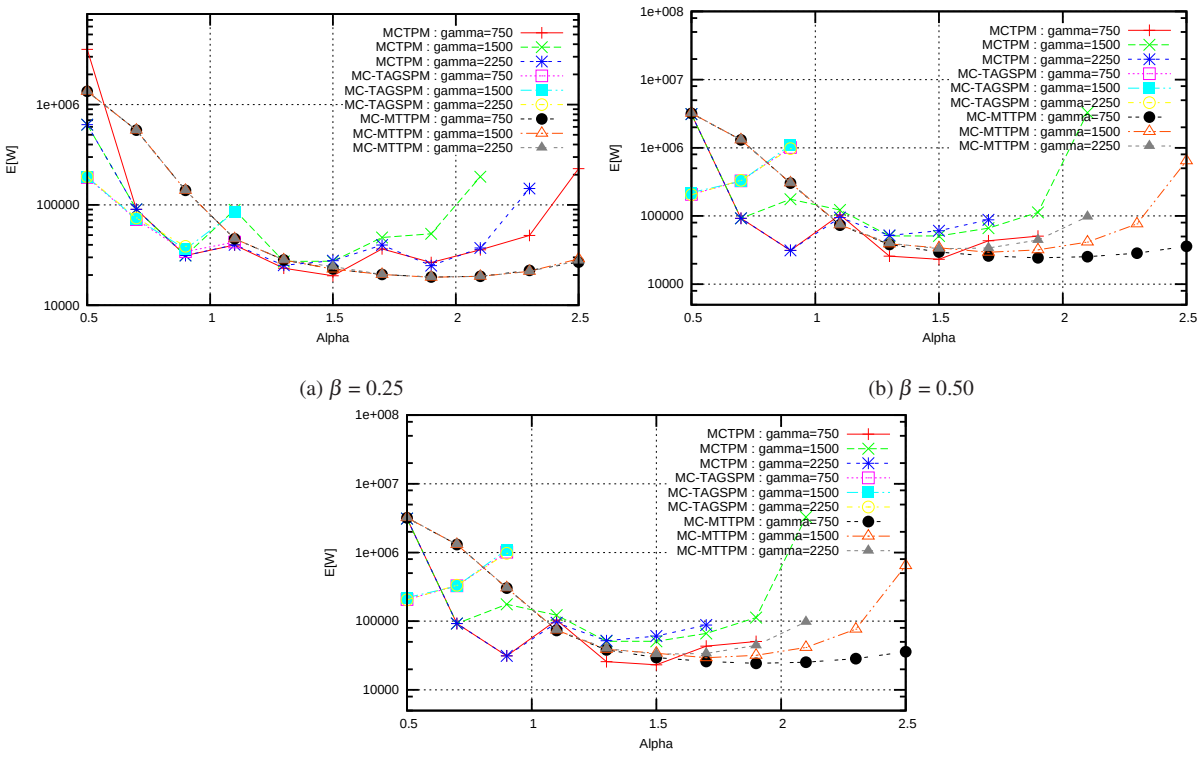
There are two main reasons why MCTPM performs so well compared to MC-MTTPM: 1) the fact that it can control the traffic flow into farms in an optimal way based on the properties of the service time distribution and the average arrival rate and 2) the fact that it supports preemptive task migration between server farms ensures that MCTPM can reduce the variability of tasks sizes in host queues significantly by migrating large tasks to other server farms. Note that the variability of task sizes in host queues is related to the performance.

Let us now briefly compare the performance of MC-Random with other three policies. MC-Random has the worst performance compared to other three policies except when α is near 2.5 and the fixed migration cost is greater than 50%. For example, under a system load of 0.3, when $\alpha = 0.5$, β = 0.25 and γ = 750, MCTPM outperforms MC-Random by a factor of 190. When α is in the proximity of 2.5, MC-Random outperforms MCTPM. The factor of improvement, however, does not exceed 2 even if the migration cost is very high (i.e. β = 0.75, γ = 2250).

5.2.4.2 High system loads

Under high system loads, the expected waiting time is highly fluctuating with increasing α (see Figure 5.10). We note that MCTPM outperforms MC-MTTPM if α lies in the range 0.5 - 1.5. As α increases beyond 1.5, MCTPM performance tends to degrade compared to MC-MTTPM. If α is near 2.5, MC-MTTPM outperforms MCTPM. In fact, if α is greater than 2.0, Random has the best

[9] It is important to point out that MC-TAGSPM is unstable under a wide range of scenarios due to very high migration cost. Under these scenarios, we cannot obtain the expected waiting time for MC-TAGSPM because we cannot compute the cut-offs for servers such that (true) system loads on all hosts are less than 1.



(a) $\beta = 0.25$

(b) $\beta = 0.50$

(c) $\beta = 0.75$

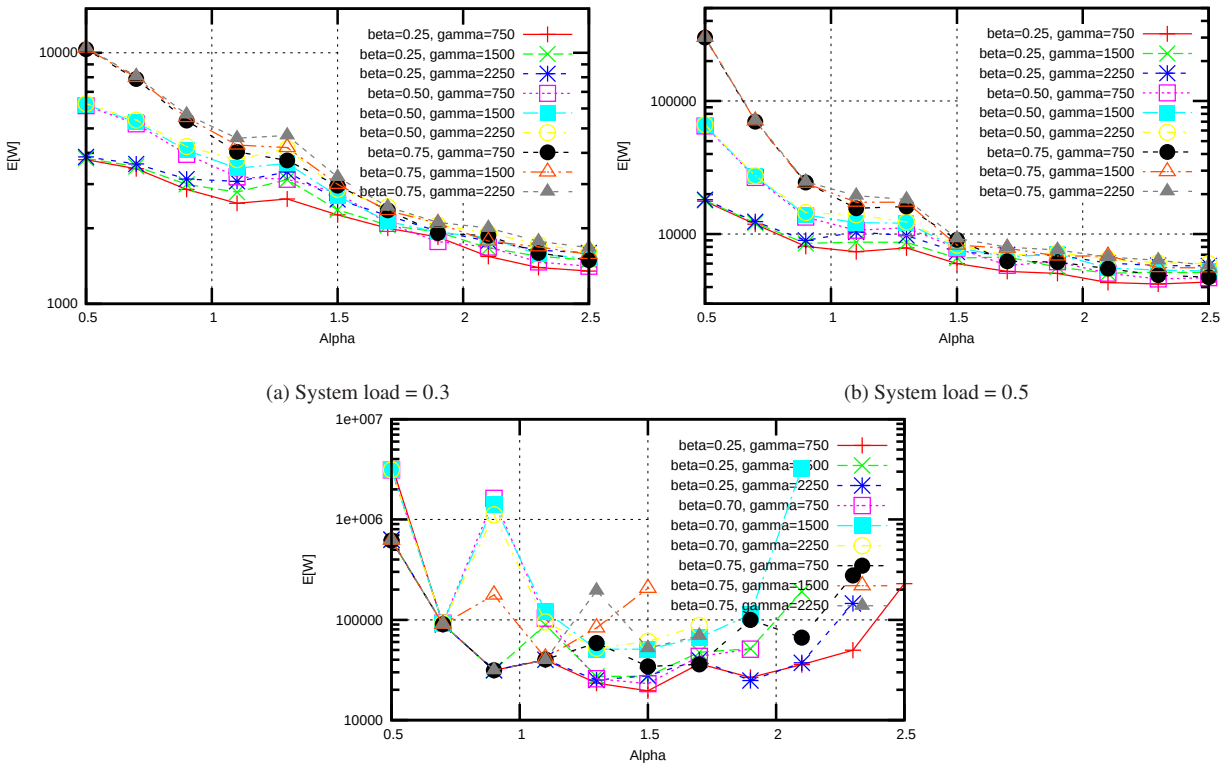
Figure 5.10: Expected waiting time for MCTPM small-sized server farms: system load = 0.7

performance under high system loads under all migration criteria. Under high system loads, the factor of improvement in MC-Random over other policies can be highly significant, if the migration cost is also high.

We note that MC-TAGSPM can operate under steady state only if α lies in the range 0.5 - 0.9. Finally, we note that when α is near 0.5, MC-TAGSPM has slightly better performance compared to MCTPM under specific migration criteria. Under all other conditions, MCTPM outperforms MC-TAGSPM.

5.2.5 Impact of migration cost on the performance of MCTPM

Preemptive task migration can result in significant performance degradations, particularly when both the proportional and fixed migration cost are high. Here we investigate the effect of migration cost on the performance of policies. Figures 5.11-a, 5.11-b and 5.11-c plot the expected waiting vs α under the system load of 0.3, 0.5 and 0.7 respectively.



(a) System load = 0.3

(b) System load = 0.5

(c) System load = 0.7

Figure 5.11: Effect of migration cost on the performance of MCTPM in small-sized server farms

Let us first consider the impact of proportional migration cost on the expected waiting time. Let r_1 be the ratio between expected waiting time of a task, when $\beta = 0.5$ and $\beta = 0.3$, and let r_2 be the expected waiting time of a task when $\beta = 0.7$ and β=0.3. Table 5.3 illustrates the values obtained for r_1 and r_2 under the system load of 0.5. We note that both r_1 are r_2 are high under low α values, which indicates that the effect of the proportional cost on the expected waiting time is significant if α is low. As α increases, both r_1 and r_2 decrease and as α approaches 2.5, the two ratios approach toward 1.

Let R_1 be the ratio between expected waiting time of a task, when $\gamma = 1500$ and when $\gamma = 750$, and let R_2 be the ratio between expected waiting time of a task when $\gamma = 2250$ and when $\gamma = 750$. Table 5.4 illustrates the values of R_1 and R_2 under the system load of 0.5. We note that both R_1 and R_2 lie in range $1.0 - 1.4$. This means that the fixed migration cost does not affect the expected waiting time at large, particularly if α is low. Similar behaviours are observed under low and high system loads.

Table 5.3: Impact of proportional migration cost on the performance: system load = 0.5.

α	r_1 γ=750	r_1 γ=1500	r_1 γ=2250	r_2 γ=750	r_2 $\gamma = 1500$	r_2 $\gamma = 2250$
0.5	3.68	3.67	3.66	16.94	16.81	16.7
0.7	2.25	2.23	2.22	5.89	5.84	5.77
0.9	1.67	1.64	1.62	3.02	2.83	2.75
1.1	1.45	1.4	1.35	2.12	2	1.88
1.3	1.42	1.39	1.26	2.04	2	1.87
1.5	1.3	1.19	1.12	1.51	1.25	1.27
1.7	1.11	1.01	0.99	1.19	1.17	1.14
1.9	1.22	1.25	1.05	1.22	1.21	1.09
2.1	1.1	1.09	1.07	1.27	1.35	1.11
2.3	1.09	1.04	1.05	1.17	1.08	1.1
2.5	1.08	1.02	1.04	1.09	1.08	1.01

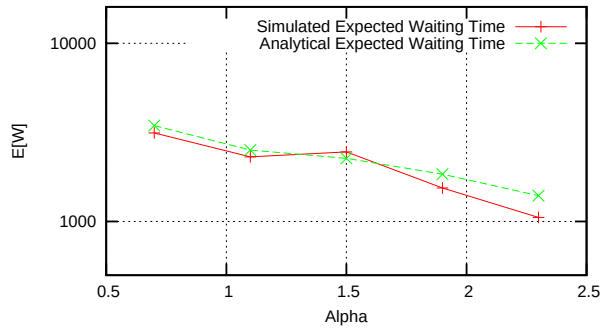
Table 5.4: Impact of fixed migration cost on the performance: system load = 0.5.

α	R_1 β=0.25	R_1 β=0.50	R_1 β=0.75	R_2 β=0.25	R_2 β=0.50	R_2 β=0.75
0.5	1.01	1.01	1.01	1.02	1.0	1.01
0.7	1.02	1.02	1.02	1.03	1.01	1.01
0.9	1.05	1.06	1.04	1.08	0.99	1.03
1.1	1.18	1.19	1.14	1.31	1.11	1.12
1.3	1.1	1.12	1.08	1.11	1.08	1.05
1.5	1.1	1.11	1.01	1.05	0.91	1.12
1.7	1.28	1.04	1.16	1.19	1.25	1.02
1.9	1.11	1.23	1.14	1.17	1.11	1.11
2.1	1.18	1.19	1.1	1.28	1.25	0.98
2.3	1.23	1.13	1.17	1.32	1.14	1.14
2.5	1.18	1.33	1.12	1.28	1.18	1.23

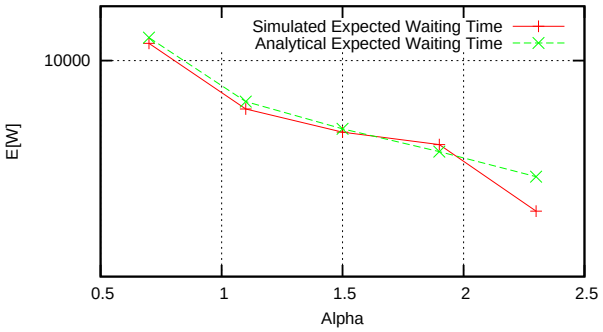
5.2.6 Simulation results

This section presents some simulation results for MCTPM. Our aim is to prove the accuracy of the analytical model presented in Section 5.2.2 by comparing the analytical results with the simulation results. We design a simulation model using C++ based OMNET++ discrete event simulator [Varg, 2001]. We generate service times and inter-arrival times using the inverse transform method (a common technique used for generate traffic from various distributions) and existing random number generators available in OMNET++. We carry out our simulations using $600,000$ tasks. The first $100,000$ in the simulation is used as a warm-up phase. It is important to point out that in order to obtain accurate results (under heavy-tailed service time distributions), we need to use a large number of tasks and a lengthy warm-up phase in our simulations.

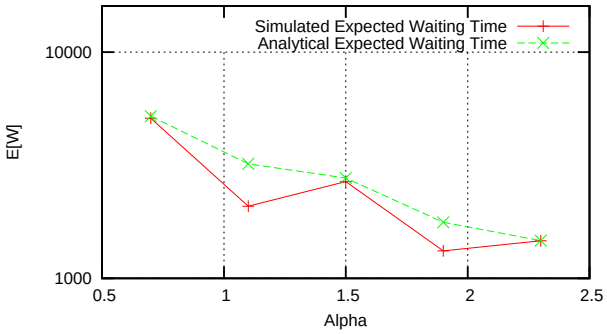
Figure 5.12 shows the simulation results and corresponding analytical results under a range of scenarios. We note that the simulation results match well with the analytical results for all the sce-



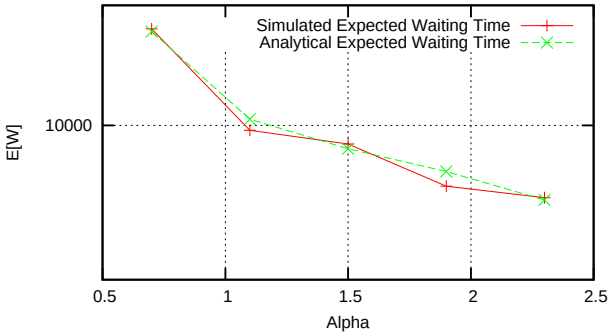
(a) System load = 0.3, $\beta = 0.25$, $\gamma = 750$



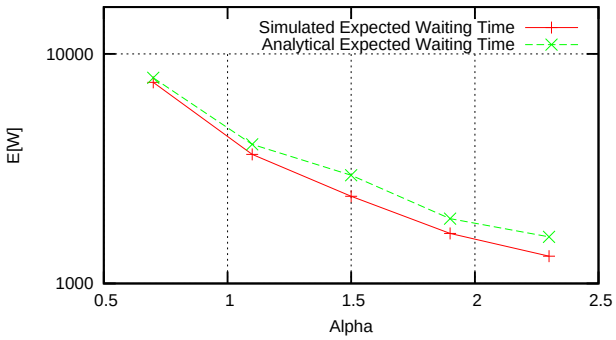
(b) System load = 0.5, $\beta = 0.25$, $\gamma = 750$



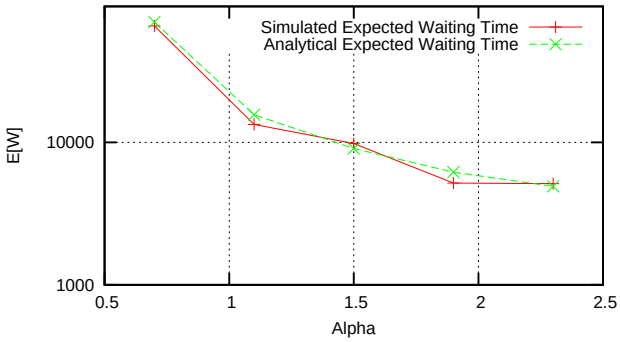
(a) System load = 0.3, $\beta = 0.50$, $\gamma = 750$



(b) System load = 0.5 , $\beta = 0.50$, $\gamma = 750$



(a) System load = 0.3, $\beta = 0.75$, $\gamma = 750$



(b) System load = 0.5, $\beta = 0.75$, $\gamma = 750$

Figure 5.12: Simulation results for MCTPM in small-sized server farms

narios considered. For example, under a system load of 0.5, when $\alpha = 0.7$, $\beta = 0.25$ and $\gamma = 750$, the simulated expected waiting time and the analytical expected waiting time are 11375.2 and 11871.61 respectively. We also note that for most of the cases, simulation results are slightly less than the

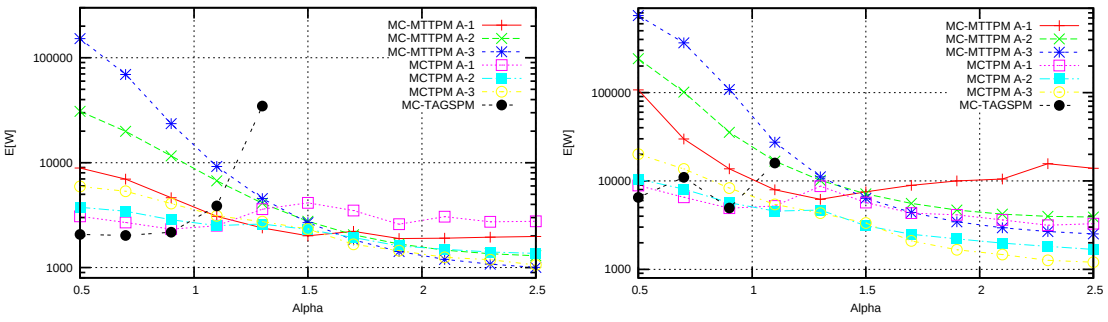
analytical results. This has been previously observed with regard to similar other task assignment policies [Harchol-Balter, 2002; Broberg et al., 2006].

5.2.7 Expected waiting time in large-sized server farms

The expected waiting for MCTPM in large-sized server farms is investigated in this section. This is important as it allows us to determine under which conditions MCTPM is more suitable for assigning tasks in large-sized server farms. First we investigate the performance of MCTPM in a 2 server farm system, where each server farm consisting of six hosts. Then, we derive the expected waiting time for MCTPM in a 2 server farm system that consists of $6i$ number of hosts, where $i = 1, 2, 3, etc$. When the number of hosts is equal to 6, there exist three possible host architectures for both MCTPM and MC-MTTPM:

- A-1: three hosts in Tier 1, two hosts in Tier 2, one host in Tier 1,
- A-2: four hosts in Tier 1, two hosts in Tier 2,
- A-3: five hosts in Tier 1, one host in Tier 1.

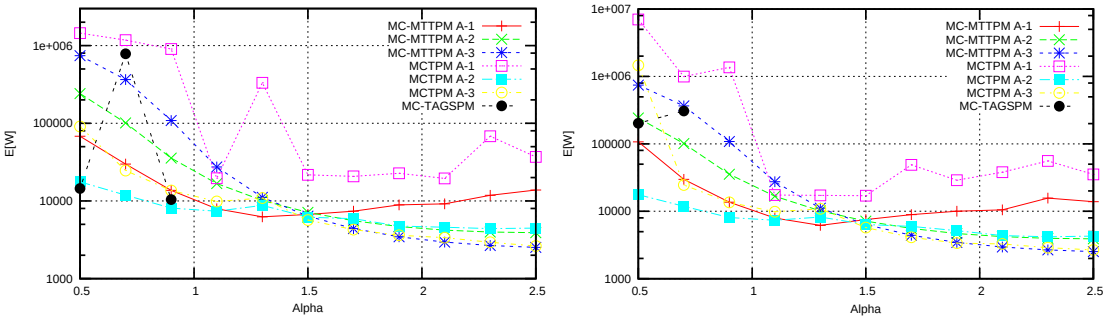
Note that the number of hosts in Tier 1 is greater than the number of hosts in Tier 2 and the number of hosts in Tier 2 is greater than the number of hosts in Tier 3. The reason for imposing this rule was previously discussed in Section 5.1. For the sake of brevity, we only consider two main migration scenarios, namely, the moderate migration cost (β = 0.25, γ = 750) and very high migration cost (β = 0.75, γ = 2250). Figures 5.13 and 5.14 illustrate the results obtained under the system loads of 0.3 and 0.5.



(a) System load = 0.3, moderate migration cost

(b) System load = 0.3, very high migration cost

Figure 5.13: Expected waiting time for MCTPM in large-sized farms: system load = 0.3



(a) System load = 0.5, moderate migration cost

(b) System load = 0.5, very high migration cost

Figure 5.14: Expected waiting time for MCTPM in large-sized farms: system load = 0.5

Notice from Figures 5.13 and 5.14 that the performance under MCTPM and MC-MTTPM highly dependent on the architecture being used. Table 5.5 illustrates the best configuration under a given set of conditions. We note that MCTPM outperforms other policies under a wide range of scenarios. When compared to MC-MTTPM, MCTPM performs significantly better if α is in the range $0.5 - 1.1$. For example, under a system load of 0.5, when $\alpha = 0.5$ and the migration cost is very high, MCTPM outperforms MC-MTTPM by a factor of 6. If α is greater than 1.1, the factor of improvement in MCTPM over MC-MTTPM is relatively small. Also note from Table 5.5 that when α approaches 2.5, MC-MTTPM outperforms MCTPM. The factor of improvement, however, does not exceed 1.2.

Table 5.5: Best task assignment policy

α	ρ = 0.3, β = 0.25, γ = 750 (moderate migration cost)	ρ = 0.3, β = 0.75, γ = 2250 (very high migration cost)	ρ = 0.5, β = 0.25, γ = 750 (moderate migration cost)	ρ = 0.5, β = 0.25, γ = 750 (very migration high cost)
0.5	MC-TAGSPM	MCTPM A-1	MC-TAGSPM	MCTPM A-2
0.7	MCTPM A-1	MCTPM A-1	MCTPM A-2	MCTPM A-2
0.9	MCTPM A-1	MCTPM A-1	MCTPM A-2	MCTPM A-2
1.1	MCTPM A-1	MCTPM A-2	MCTPM A-2	MCTPM A-2
1.3	MCTPM A-2	MCTPM A-2	MCTPM A-2	MCTPM A-2
1.5	MCTPM A-2	MCTPM A-2	MCTPM A-3	MCTPM A-3
1.7	MCTPM A-3	MCTPM A-3	MCTPM A-3	MCTPM A-3
1.9	MC-MTTPM A-3	MCTPM A-3	MC-MTTPM A-3	MCTPM A-3
2.1	MC-MTTPM A-3	MCTPM A-3	MC-MTTPM A-3	MC-MTTPM A-3
2.3	MC-MTTPM A-3	MCTPM A-3	MC-MTTPM A-3	MC-MTTPM A-3
2.5	MC-MTTPM A-3	Random	Random	MC-MTTPM A-3

We now consider the case where there are more than six hosts in one server farm. Let us consider the following three host architectures:

- A-1*: $3i$ hosts in Tier 1 $2i$ hosts in Tier 2 and i hosts in Tier 3,
- A-2*: $4i$ hosts in Tier 1 and $2i$ hosts Tier 2,
- A-3*: $5i$ hosts in Tier 1 and i hosts in Tier 2,

where $i = 2,3,4,etc$. Note that the total number of hosts in a farm is equal to $6i$ $(= 3i+2i+i = 4i+2i = 5i+i)$. It is important to point out that several other host architectures are also possible, particularly for large i. However, we only consider these three because we can derive the expected waiting time for these three architectures from the results, which we obtained for the case of six hosts. From the performance model presented in Section 5.2.2, we notice that under a given scenario (i.e. system load and α), the expected waiting time for MCTPM under A-1* is the same as the expected waiting time for MCTPM under A-1 because both A-1 and A-1* possess the same parameters. Similarly, the expected waiting time for MCTPM under A-2* is equal to the expected waiting time for MCTPM under A-2 and so on. Therefore, we can claim that if the number of servers in a farm is equal to $6i$, where $i = 2,3,4,etc.$, MCTPM can perform at least as good as a 6 Host system. When $i > 1$, several other architectures are possible and it is likely that some of these architectures can provide better performance compared to A-1*, A-2* and A-3*.

Let us now consider the case where there are more than two server farms. Unfortunately, we are not able to present the analytical results for more than two farms because we cannot solve the optimisation problems for more than two server farms using the resources available to us. However, in this case it is possible to configure these farms to produce at least as good as the best performance of two server farms. For example, in the case of four farms, we can have two MCTPM systems, where each MCTPM consists of two farms and tasks can be dispatched to each MCTPM system with an equal probability.

5.3 Conclusion

This chapter proposed an efficient task assigning policy (MTTMEL) for assigning tasks in stand-alone batch server farms. We showed that MTTMEL outperforms existing policies under a range of conditions. This chapter also proposed MCTPM for assigning tasks in multiple server farms. MCTPM exploits the properties in multiple server farm environments and therefore, it performs better compared to those that optimise the performance in stand-alone farms. MCTPM was based on a multi-tier host architecture that supports preemptive task migration. The proposed model controls the traffic flow into server farms so as to optimise the performance. The flexible multi-tier host architecture of the proposed policy significantly reduces the variance of task sizes in host queues.

We developed an analytical model for the proposed model in which we incorporated the migration cost. In Section 5.2.3 we showed that the proposed policy outperforms existing policies under a wide range of workload conditions. In the specific cases where MCTPM did not outperform other policies, its performance does not degrade significantly except under the specific case, where the system load, migration costs and α are high, and the number of servers in the farm is small. In large-sized server farms, this specific case can be avoided by changing the number of tiers and the number of hosts allocated to tiers.

Chapter 6

ADAPT-POLICY: Task Assignment in Distributed Systems when the Service Time Distribution of Tasks is not known *a Priori*

The task assignment policies proposed in the previous chapters were related to assigning tasks in server farms under heavy-tailed service time distributions. The particular focus on heavy-tailed service time distributions is due to the fact that service times of majority of tasks that appear in computing environments closely follow heavy-tailed distributions. However, as pointed out in Section 2.1.2, heavy-tailed service time distributions cannot be justified for certain tasks that appear in certain environments [Zhang and Sun, 2005; Riska et al., 2002]. There are two main reasons for this: 1) service times of certain tasks are not always recorded. As such, it is difficult to find a sufficient number of data sets, which can be used to fit a probability distribution to the entire population of task sizes and 2) even if such data sets are available, they may come from heterogeneous family of distributions and any attempt to fit a particular distribution to it would be impossible. Moreover, there is also a possibility for the service time distribution of tasks to vary over time due to the non-stationary nature of traffic [j. Lin et al., 2006; Bertsimas and Mourtzinou, 1997; Zhang et al., 2003].

This chapter investigates a way to design task assignment policies to efficiently assign tasks in server farms under varying (i.e. non-stationary) traffic conditions when the service time distribution of tasks is not known *a priori*. Much existing tasks assignment policies [Harchol-Balter et al., 1999; Broberg et al., 2004; 2006; Harchol-Balter, 2002; Tari et al., 2005; Zhang and Fan, 2008], including

those presented in the previous chapters, are static by nature because they are based on a particular stationary service time distribution. In other words, these policies assume particular (parametric) service time distribution (e.g. Pareto, Exponential, etc.) and a set of fixed (static) parameters for that particular parametric distribution. Due to such assumptions, they can compute their optimal scheduling parameters (server cut-offs, etc.) off-line so as to optimise a given performance criteria such as the expected waiting time, expected slowdown and expected load. Since the scheduling parameters for these policies are computed off-line to optimise the performance under a specific scenario (e.g. Pareto distribution with $\alpha = 0.5$), they [Harchol-Balter, 2002; Broberg et al., 2006; 2004; Harchol-Balter et al., 1999] cannot respond to variations that occur in the incoming service time distribution. As such, the performance of these policies degrades under constantly evolving operational conditions.

This chapter proposes an adaptive task assignment policy, called ADAPT-POLICY, which is based on the concept of multiple task assignment policies. ADAPT-POLICY defines a set of static-based task assignment policies for a given distributed system taking into account the specific properties of the system. These policies are selected in such a way that they have different performance characteristics under different workload conditions (e.g. service time distributions, etc.). The objective is to use the policy with the best performance (i.e. the one with the least expected waiting time) to assign tasks. Which policy performs the best depends on the traffic conditions that vary over time. ADAPT-POLICY determines the best task assignment using the service time distribution of tasks (and various other traffic properties), which is estimated on-line and then it adaptively changes the task assignment policy to suit the most recent traffic conditions. Since ADAPT-POLICY has the ability to change the task assignment policy, it performs well compared to the static-based task assignment policies, which optimises the performance under a particular workload scenario.

There exist a handful of task assignment policies, which are of similar nature to ADAPT-POLICY. Two such important policies are ADAPTLOAD [Zhang and Sun, 2005] and EQUILOAD [Ciardo et al., 2001]. Unfortunately, these two policies have the following issues.

- Both EQUILOAD and ADAPTLOAD assume that the processing requirements of tasks can be estimated prior to execution of tasks. As such, these policies cannot be used to assign dynamic content and majority of scientific workloads. Recall that one of the main aims of this thesis is to design policies that can assign dynamic content and scientific workloads whose service times are not known *a priori*.

- ADAPTLOAD approximates the service time distribution of tasks using the discrete histogram of service times. Using this discrete histogram, it obtains the probabilities needed for comput-

ing scheduling parameters (e.g. server cut-offs, fractions of tasks assigned to hosts, etc.) for the system. Unfortunately, these probabilities are not accurate because the discrete histogram does not accurately capture the true characteristics of the actual service time distribution, particularly when the service time distribution has long-tails and rapid fluctuations [Ciardo et al., 2001]. This often results in incorrect scheduling decisions, which results in significant performance degradations.

- EQUILOAD estimates the service time distribution of tasks off-line by fitting the service times of tasks into a hyper-exponential or a hypo-exponential distribution. Using this estimated distribution, EQUILOAD computes the scheduling parameters (e.g. server cut-offs, etc.) for the system. The main issue with this approach is that it only produces accurate results if the actual service times of tasks closely follow hyper-exponential or hypo-exponential distributions. For other types of service time distributions, the performance of EQUILOAD can be very poor because the estimated service time distribution significantly deviates from the true service time distribution. The other main issue with EQUILOAD is that it cannot respond the variations that occur in the incoming service time distribution as the distribution fitting is carried out off-line.

- Both ADAPTLOAD and EQUILOAD attempt to equalise the expected load at hosts and this is relatively easy to achieve compared to that of optimising certain performance metric such as the expected slowdown and expected waiting time. However, equalising the expected load does not always improve the performance [Crovella et al., 1998a].

The core features of ADAPT-POLICY include:

- On-line data collection: Unlike EQUILOAD and ADAPTLOAD, ADAPT-POLICY does not make any assumptions regarding the actual sizes of tasks and as such, it has no means of knowing the service times prior to execution. For each task that completes its processing, ADAPT-POLICY computes its service time by subtracting the arrival time from the departure time. ADAPT-POLICY stores these processing times at individual hosts in their main memory or in a form of file. After the system completes processing n number of requests the density estimation is performed.

- On-line density estimation using non-parametric based techniques: Unlike EQUILOAD, ADAPT-POLICY does not fit the service times of tasks into a parametric distribution (e.g. hyper exponential), rather it estimates the service time distribution of tasks and various distributional properties (e.g. variance) on-line using non-parametric distribution estimation techniques [Wand and Jones, 1995]. Non-parametric techniques impose fewer restrictions on the

underlying probability distributions. As such, they are considered to be more flexible methods of estimation with a wider range of validity compared to parametric methods of estimation (such as Method of Moments and Maximum Likelihood Estimation [Mood et al., 1974]).

- On-line selection of task assignment policies: Different systems support different policies and for many such systems, it is possible to define a set task assignment policies that have different performance characteristics under different traffic conditions. ADAPT-POLICY defines a set of static-based policies for a given distributed system based on its properties, and later uses the task assignment policy with the least expected waiting time for assigning the next batch of tasks.[1] The task assignment policy with the least expected waiting time is determined using the service time distribution and various distributional properties of the service time distribution that are estimated on-line.

The rest of this chapter is organised as follows. Section 6.1 presents the details of ADAPT-POLICY. Section 6.2 evaluates the performance of the proposed non-parametric density estimator followed by an experimental performance analysis of ADAPT-POLICY under various workload scenarios is presented in Section 6.3. The chapter is concluded in Section 6.4.

6.1 ADAPT-POLICY

ADAPT-POLICY consists of three main stages, namely, the data collection, density estimation and selection of task assignment policies. In the data collection stage, each host records service times of tasks. These service times are used in the density estimation stage to estimate the probability distribution function, cumulative distribution of service times and various other distributional properties of the service time distribution. In the policy selection stage, ADAPT-POLICY defines a set of static-based task assignment policies for a given system and it utilises the policy with the least expected waiting time for assigning the next batch of tasks.

6.1.1 On-line data collection

The aim of on-line data collection is to collect the service times of tasks, which are used in the density estimation stage to estimate various distributional properties of the service time distribution. The data collection stage is illustrated in Figure 6.1. For each new task arriving at the dispatcher, it assigns an

[1] In the very rare case of a given distributed system only supporting one particular task assignment policy, ADAPT-POLICY can still be of benefit to that system as it can be used to adjust the scheduling parameter of that policy according to changes that are occurring in the incoming traffic stream.

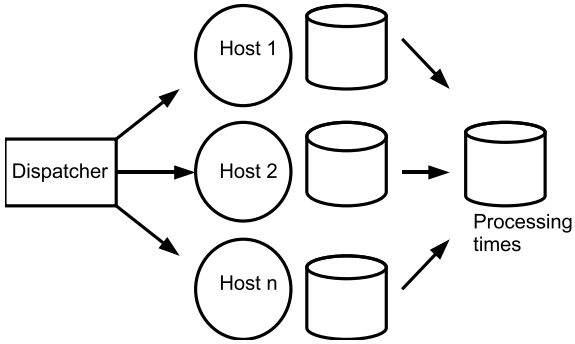


Figure 6.1: ADAPT-POLICY: data collection stage

ID and records the task arrival time. These values are stored within the task. In the case of HTTP requests, such values can be stored in the HTTP header itself. Each incoming task is then assigned to a host using Policy P_L, where P_L denotes the task assignment policy with the least expected waiting time. Further details about the way to select task assignment policies is discussed in Section 6.1.3.

For each task that completes processing at a given host, ADAPT-POLICY computes its service time by subtracting the arrival time from the departure time. ADAPT-POLICY stores these service times at individual hosts in their main memory or in the physical memory in a form of file. After the system completes processing n tasks the density estimation is carried out using these service times (this will be discussed in detail in Section 6.1.2). Note that the task IDs can be used to track the total number of tasks processed in the system. The value of n needs to be relatively high in order to capture the rapid changes and heavy-tails (e.g. Pareto distributions with low α values) [Harchol-Balter, 2002] that are common in real traffic.

6.1.2 On-line density estimation using non-parametric based techniques

This section outlines the second stage of ADAPT-POLICY, where it estimates the probability density function, cumulative distribution function and moments of the probability distribution function. Figure 6.2 illustrates the main steps involved in this stage. Recall that the key idea behind ADAPT-POLICY is to use the policy with the least expected waiting time to assign the next batch of requests. The policy with the least expected waiting time is computed using the probability density function, cumulative density function, moments of the service time distribution and average arrival rate of tasks that are estimated on-line. ADAPT-POLICY uses non-parametric kernel based density estima-

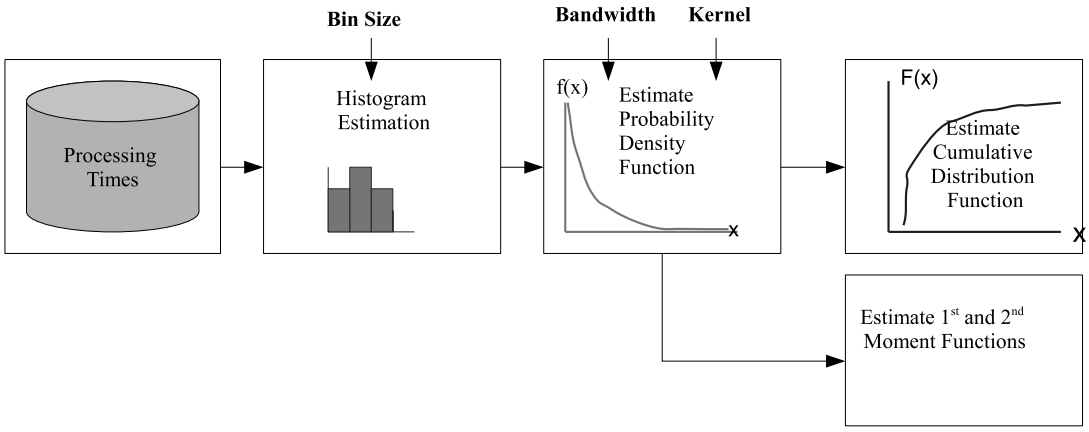


Figure 6.2: ADAPT-POLICY: density estimation stage

tion techniques [Wand and Jones, 1995] to estimate these distributions and moments. Non-parametric techniques have advantages by not imposing many restrictions on the underlying probability distributions. Therefore, they are considered a more general approach to estimation with a wider range of validity compared to parametric methods of estimation [Wand and Jones, 1995]. Let us now discuss the main components of non-parametric density estimation related to ADAPT-POLICY.

(a) Estimation of probability density function (of service times)

Let us now present the model related to the estimation of the probability density function (PDF) of service times. Let X be a univariate continuous random variable distributed according to probability density f, where f is unknown. For any set B of real numbers, the probability that X belongs to this set is given by

$$P(X \in B) = \int_B f(x)dx. \tag{6.1}$$

For example, when $B = [a,b]$, $P(a \leq X \leq b) = \int_a^b f(x)dx$. The aim is to estimate $f(x)$ using sample observations $(X_1, X_2, ...Xn)$ of service times.[2] The estimator of $f(x)$ (probability density function) at $x = x_0$ is given by

$$\hat{f}(x_o) = \frac{1}{nh_n} \sum_{i=1}^{n} K\left(\frac{x_o - X_i}{h_n}\right), \tag{6.2}$$

where h_n and $K(.)$ are the bandwidth and kernel function respectively. The bandwidth plays a major role in kernel density estimation and its main task is to control the amount of smoothing given to

[2]Note that the probability density function completely characterises the 'behaviour' of a random variable.

the density estimator. A small bandwidth will result in observations closest to the sampled point, x_0, receiving more weight, whereas a large bandwidth will over–smooth the density estimator resulting in the fit assuming an undulatory appearance. Later in this section, we will provide the details of various bandwidth selection methods and kernel functions.

(b) Transformation-based kernel estimator

Extensive numerical experiments using numerous probability density functions indicate that the kernel density estimator defined by Equation 6.2 often performs poorly for highly skewed data sets that exhibit very high variability (e.g. Pareto distributions with low α values). To address this problem, we adapt the transformation-based kernel estimation, where we transform the data set (i.e. service times) into a different scale prior to estimating the probability density function [Simonoff, 1996]. The transformed distribution is later transformed back to its original domain. Let $f(x)$ denotes the probability density function in the original domain and let $y = T(x)$ be the transformation of x into y. Assume that $T(x)$ is monotonic, i.e. strictly increasing or decreasing and let $T^{-1}(x)$ be the unique inverse of $T(x)$. Then, the transformed probability density function $f_T(y)$ in the transformed domain is related to $f(x)$ by

$$f_T(x) = f(T^{-1}(x))\left|\frac{dT^{-1}}{dx}(x)\right|. \tag{6.3}$$

The transformation-based kernel estimator is given by

$$\hat{f_T}(x_0) = \frac{1}{nh}\left|\frac{dT^{-1}}{dx}(x)\right|\sum_{i=1}^{n} K\left(\frac{[T(x_o) - T(x_i)]}{h}\right). \tag{6.4}$$

The transformation we use is the exponential transformation, i.e. $T(x) = e^x$. Since $T^{-1}(x) = ln(x)$ and $\frac{dT^{-1}}{dx}(x) = \frac{1}{x}$, the transformed-based kernel estimator is given by

$$\hat{f_T}(x_0) = \frac{1}{xnh}\sum_{i=1}^{n} K\left(\frac{[ln(x_0) - ln(x_i)]}{h}\right). \tag{6.5}$$

If the service time distribution of tasks is not highly skewed or highly variable, the use of transformation-based kernel is not necessary. However, our experiments indicate that transformation-based kernel estimator is able to accurately estimate both skewed and non-skewed distributions (e.g. uniform) as long as the sample size is relatively large, which is the case for the data sets that we are considering in this chapter. Therefore, we use the transformation-based kernel to estimate the probability density function of service times. It is important to point out that the transformation-based estimator we employ in this chapter is one of the least complex methods proposed in the literature to estimate skewed distributions. There are other advanced methods that may give somewhat more

accurate results [Simonoff, 1996]. However, these methods are computationally very intensive, and therefore they may not be suitable for online probability density estimation [Simonoff, 1996].

(c) Bandwidth

The bandwidth (of the kernel density estimator, i.e. h_n) plays an important role in the density estimation in determining the smoothness of the estimated probability density function. If the bandwidth is too small, the estimated probability density function may become rough and reflect too closely the features of real data rather than the true density function. Larger bandwidths, on the other hand, can lead to over smoothing that can lead to the important features of the distribution function becoming less distinct. Numerous methods such as *Rules of Thumb*, *Least Squares Cross-Validation*, *Biased Cross-Validation* and *Solve-the-Equation Plug-In Approach* have been proposed as bandwidth estimation methods [Jones et al., 1996]. Unfortunately, all of these methods are computationally very intensive (e.g. $O(n^3)$) as they are associated with complex optimisation problems. As such, we do not employ them in this chapter. For more information about these methods, the reader may refer to [Jones et al., 1996]. In this chapter we use the method proposed in [Isogai, 1987] due its simplicity (not computationally intensive), its wide usage and its effectiveness. The bandwidth proposed by [Isogai, 1987] has the following form:

$$h_n = n^{-r}, \quad 0.2 < r < 1, \tag{6.6}$$

where h_n and n denote the bandwidth and size of the sample respectively. The effect of r on the results are discussed in Section 6.2.

(d) Kernel function

The next important step in the density estimation is to select a kernel function denoted by $K(.)$. The kernel function is a symmetric probability density function satisfying the following conditions.

$$\begin{aligned} &\int_{-\infty}^{\infty} K(t)dt = 1, \\ &\int_{-\infty}^{\infty} tK(t)dt = 0, \\ &\int_{-\infty}^{\infty} t^2K(t)dt = k_2 < \infty. \end{aligned} \tag{6.7}$$

The first two conditions indicate that the kernel function is a symmetric probability density function, while the last condition states that the second moment of the kernel function is always finite. Some widely used kernel functions are Epanechnikov [Wand and Jones, 1995], Triangular [Wand and Jones, 1995], Normal kernels [Wand and Jones, 1995] and Biweight [Wand and Jones, 1995]. It is rather difficult to favour one kernel function over the other as the efficiency of different kernels differ under

different efficiency measures. Efficiency measures used to evaluate effectiveness of kernel functions include the mean integrated squared error (MISE) and integrated standard error (ISE), MISE being the most widely used method. The MISE is computed as

$$MISE(n) = E[\int |f_n(x) - f(x)|^2 dx], \tag{6.8}$$

where E denotes the expected value. f and f_n denote the unknown probability density function and the estimated probability density function respectively. They are based on a sample of n identically and independently distributed (iid) random variables. Most of the popular kernels functions have very small MISE values. In ADAPT-POLICY, we employ the Epanechnikov kernel [Silverman, 1986; Wand and Jones, 1995] whose kernel function is given by

$$K(t) = \frac{3}{4}(1-t^2), \qquad |t| \leq 1. \tag{6.9}$$

Now that we have set on the kernel density estimator, the kernel function and the bandwidth selection method, let us now discuss the way to estimate the probability density function of the service times using the kernel estimator given by Equation 6.5. The main steps involved in the density estimation are as follows.

- Apply the transformation-based general density estimator (given by Equation 6.5) to the service times of tasks and estimate the $\hat{f}(x_i)$ (PDF) at different x_i values.
- Store the estimated $\hat{f}(x_i)$ and (corresponding) x_i values in an array of points, where each point consists of two fields, namely, the value of the random variable (x_i) and the estimated value $\hat{f}(x_i)$. The data structure for the density array is illustrated in Figure 6.3.

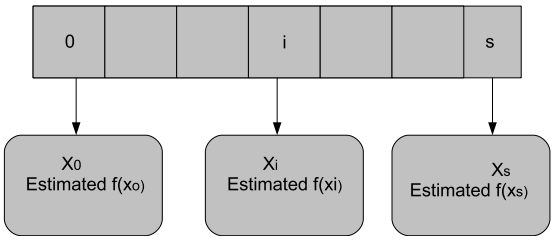


Figure 6.3: ADAPT-POLICY: density array

The simplest way to populate this array is to compute $f(x_i)$ using equally and regularly spaced values of x_i. This results in more accurate results, but it requires a large number of computations to populate

the data structure which is very large in size. Small array sizes with equally spaced x_i values, on the other hand, may not produce very accurate results because they may not accurately capture the behaviour of the density functions with long-tails and rapid fluctuations due to under-sampling.[3] However, even with a small number of x_i values (i.e. small array sizes), it is possible to achieve high accuracy if we position these x_i values based on the characteristics of the distribution being estimated. We propose a new method to populate the array, which is based on the histogram of service times.[4] The criteria for populating the density array are as follows.

- Construct the histogram using the service times of tasks. This requires us to specify the number of bins. Let N_{bin} be the number of bins and n be the number of tasks in the sample. The number of bins for a histogram is computed using the following standard formula:

$$N_{bin} = \sqrt{n}. \tag{6.10}$$

- Compute the distance between two consecutive x_i values in the density array. Let δx_i be the distance between two consecutive x_i values that belongs to bin i. Then,

$$\delta x_i = \begin{cases} \frac{Task[H[i]-1]-Task[H[i-1]-1]}{\frac{Lh[i]}{n}}, & i > 1, \\ \frac{Task[H[i]-1]-Task[H[1]}{\frac{Lh[i]}{n}}, & i = 1, \end{cases} \tag{6.11}$$

  where L, $h[i]$ and $H[i]$ are the length of density array (specified by the user), the frequency of bin i (computed using the histogram) and the cumulative frequency of bin i (computed using the histogram) respectively. $Task[i]$ denotes the service time of i^{th} task when the tasks are arranged in an ascending order of their size. By computing δx_i according to Equation 6.11, we ensure that the sufficient number of x_i values are used in the regions, where there is a high concentration of data. This guarantees that we are accurately capturing the rapid fluctuations that may exist in these regions.

(e) Cumulative distribution function (CDF) estimation

The previous section presented the equations and the data structures relating to estimation of the probability density function of service times. This section provides the equations and data structures associated with the estimation of the cumulative distribution function (CDF) of service times. As discussed, the cumulative distribution function of service times is required to compute the expected waiting time for tasks assignment policies in the task assignment policy list of ADAPT-POLICY.

[3] Note that the size of the density array is not same as the number of tasks in the sample.

[4] The histogram is the simplest form of probability density estimation, where each bar in the histogram represents the number of occurrences (frequency) of the random variable within a disjoint range called the bin.

Recall that ADAPT-POLICY aims at using the task assignment policy list with the least expected waiting time for assigning the next batch of tasks.

The cumulative distribution function of a continuous probability density function can be obtained by integrating the probability density function between the lower and upper limits as follows:

$$F(x) = \int_0^{\infty} f(t)dt, \tag{6.12}$$

where $F(x)$ is the cumulative distribution function of $f(x)$. Since the probability density function we estimated in the previous section is stored in an array in a form of discrete values, we cannot use integration to obtain $F(x)$. This means that we need to employ numerical integration techniques such as Riemann sum [Epperson, 2001], Simpson's rule or Trapezoid rule. This chapter utilises the Riemann sum (See Figure 6.4) to approximate the cumulative distribution function. The Riemann sum is mathematically less complex compared to Simpson's rule, which requires fitting of quadratic equations. Let us now provide the equations relating to the estimation of CDF. Let f be a real value

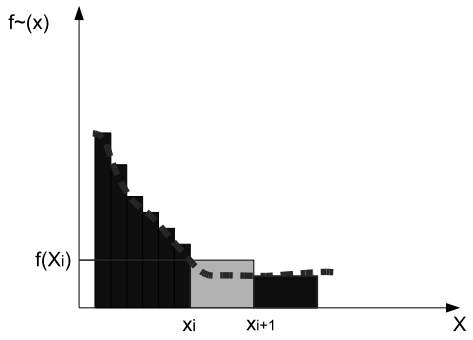


Figure 6.4: Numerical integration: Riemann Sum

function defined in the interval $[a,b]$ and let $x_o, x_1, x_2, ...x_n$ such that $a = x_0 < x_1 < x_2 ... < x_n = b$ creating a partition $P = [x_0, x_1), [x_1, x_2), ...[x_{n-1}, x_n]$ of $[a,b]$. The Riemann sum of f over $[a,b]$ with the partition p is defined as

$$R = \sum_{i=0}^{n-1} f(y_i)(x_{i+1} - x_i), \tag{6.13}$$

where $x_i \neq x_{i+1}$. y_i can be chosen arbitrarily within the interval $[x_i, x_{i+1}]$. We choose y_i such that $y_i = x_i$, i.e the left end point of the interval. Let $\hat{F}(x)$ and $\hat{f}(x)$ be the estimated CDF and PDF respectively. Then,

$$\hat{F}(x_i) = \sum_{i=0}^{n-1} \hat{f}(x_i)(x_{i+1} - x_i). \tag{6.14}$$

$\hat{F}(x)$ is evaluated at different x_i values (note: x_i values are computed according to Equation 6.11) and stored in the array data structure shown in Figure 6.3. Note that the size of this cumulative distribution array is equal to $s-1$, where s is the size of probability density function array.

(f) Estimating moments

The last step in the density estimation stage is to estimate the first and second moment of the service time distribution. These moments are used later to compute the expected waiting time of task assignment policies in the task assignment policy list of ADAPT-POLICY.

First and second moments are obtained as follows:

$$\hat{M}_1(x_i) = \sum_{i=0}^{n-1} x_i \hat{f}(x_i)(x_{i+1} - x_i), \tag{6.15}$$

$$\hat{M}_2(x_i) = \sum_{i=0}^{n-1} x_i^2 \hat{f}(x_i)(x_{i+1} - x_i). \tag{6.16}$$

By following similar steps taken in the process of estimating CDF, $\hat{M}_1(x_i)$ and $\hat{M}_2(x_i)$ are evaluated at different x_i values (according to Equation 6.11) and stored in the array data structure shown in Figure 6.3.

6.1.3 On-line selection and deployment of task assignment policies

This section discusses the last step of ADAPT-POLICY, where a set of static-based policies are introduced for a given distributed system taking into consideration the properties of the system. These task assignment policies have been selected in such a way that they have different performance characteristics under different workload conditions (i.e. service time distributions, etc.) and therefore, they can perform well under a range of workload conditions. Let us now provide a few examples of different types of systems and task assignment policies that can be utilised in these systems.

- Batch computing server farms with non-preemptive task migration facilities: Random, Round-Robin, TAGS, TAPTF and MTTMEL.
- Batch computing server farms with no task migration facilities: Random, Round-Robin, TAGS-PM, TAPTF-WC and MTTPM.
- Time sharing server farms with no task migration facilities: Random, Round-Robin and MLMS.
- Time sharing server farms with non-preemptive task migration facilities: Random, Round-Robin, MLMS and MLMS-M.

- Time sharing server farms with preemptive task migration facilities: Random, Round-Robin, MLMS and MLMS-PM.

The policies listed for each of the above systems, are optimal under different workload conditions. As such, these policies can efficiently handle a wide range of (varying) traffic conditions.

For each policy in the task assignment policy list, ADAPT-POLICY derives an (general) expression for the expected waiting time by applying queueing theory.[5] For example, the expected waiting time under Random task assignment policy, which distributes tasks among back-ends hosts with an equal probability is given by

$$E[W] = \frac{\lambda_i E[X^2]}{2(1-\lambda E[X])}, \tag{6.17}$$

where $E[X^2]$ and $E[X]$ represent 2^{nd} and 1^{st} moments of the service time distribution respectively and λ denotes the average arrival rate into Host i.[6] For advanced policies, such as TAGS and MLMS, the expected waiting time will have more complex forms. The previous two chapters discussed the way to derive the expected waiting for these advanced policies. We noted that the expected waiting time for these policies are functions of several factors such as the arrival rate into the system, probability distribution function of tasks, moments of service time distribution and scheduling parameters.[7]

Using these derived expected waiting time expressions, ADAPT-POLICY computes the expected waiting time for each task assignment policy (on-line) using the average arrival rate, estimated service time distribution and estimated distributional properties (refer to Section 6.1.2).[8] Once the expected waiting time for policies are computed, the task assignment policy with the least expected waiting time is communicated to the dispatcher, which then starts assigning tasks using that policy.

6.1.3.1 On-line optimisation

Computing the expected waiting time for task assignment policies, such as Random and Round-Robin, is straightforward because these policies do not have any scheduling parameters (e.g. server cut-offs, etc.) that need to be optimised based on the traffic properties. However, for advanced policies [Harchol-Balter et al., 1999; Harchol-Balter, 2002; Zhang and Sun, 2005; Broberg et al., 2004; 2006], this is not the case and to compute the expected waiting time for these policies, complex

[5] It is important to point out that defining the task assigning policies and deriving the expected waiting time for task assignment policies are carried out off-line.

[6] Note that under Random each host in the system sees the same processing time distribution.

[7] Scheduling parameters refers to parameters such as probability of tasks assigned to hosts and processing time ranges used for hosts [Harchol-Balter et al., 1999; Harchol-Balter, 2002; Zhang and Sun, 2005; Broberg et al., 2004; 2006]. It is important point out that not all tasks assignment policies have scheduling parameters, especially the traditional ones.

[8] The average arrival rate is computed on-line using number of task that arrive during the pervious time period.

optimisation problems need to be solved. Let us now discuss the way these optimisation problems are handled in ADAPT-POLICY. Almost all of these optimisation problems are non-linear optimisation problems. Therefore, linear optimisation techniques (such as linear programming and simplex method) cannot be used for optimisation. Moreover, certain non-linear optimisation techniques (such as quadratic programming) cannot be used as well because quadratic programming requires the objective function to be in a quadratic form which is not the case for many task assignment policies [Harchol-Balter, 2002; Broberg et al., 2004; 2006].

Evolutionary algorithms have been successfully used to in the past to solve non-linear optimisation problems [Menon, 2004; Srinivas and Patnaik, 1994]. The question relates to though the selection of a suitable evolutionary algorithm that can be incorporated with ADAPT-POLICY. Certain evolutionary algorithms are based on fundamentals of genetic evaluations, whereas some others are based on natural phenomenon (such as swarms and ants [Menon, 2004; Srinivas and Patnaik, 1994]). Genetic evolutionary algorithms suffer from the problem of slow convergence, meaning that these algorithms could take a long time to reach a near optimal solution [Srinivas and Patnaik, 1994]. Non-genetic algorithms (such as particle swarm optimisation (PSO)), on the other hand, can reach a solution much faster compared to that of genetic algorithms [Kennedy and Eberhart, 1995; Poli et al., 2007]. Moreover, PSO has shown better performance over other non-genetic algorithms such as hill climbing because PSO is based on both local and global searching techniques. In ADAPT-POLICY we utilise the basic version of PSO. This iteratively improves (optimises) its solution with respect to a given measure of quality. PSO places its particles in the search space of the objective function and the objective function is evaluated at each iteration. The movement of the particles in the search space is determined by a simple mathematical formula, which takes into account the position and the velocity of particles. It is not our intention to discuss the PSO in detail here, since we simply use it as a technique for solving the optimisation problems. More details about PSO can be found in [Kennedy and Eberhart, 1995; Poli et al., 2007].

6.2 Evaluation of density estimation

This section evaluates the performance of kernel density estimators (i.e. Equations 6.5 and 6.14) presented in Section 6.1.2. The aim is to show that the kernel density estimators can be used to estimate a range of service time distributions with different properties. We show that the density estimators presented in Section 6.1.2 can accurately estimate both PDF and CDF of a range of service time distribution and therefore, these estimators can be successfully used when the service time distribution of tasks is not known *a priori*.

We apply the kernel density estimators to estimate three different types of distributions, namely, the Bounded Pareto, Bounded exponential and Bounded Weibull. We have chosen these service time distributions to cover a range of scenarios. It is important to note that the kernel density estimator does not assume any knowledge of these service time distributions. Figures 6.5, 6.6, 6.7, 6.8, 6.9, 6.10, 6.11, 6.12 and 6.13 compare the estimated and actual distributions for Bounded Pareto, Bounded exponential and Bounded Weibull under different r (refer to Equation 6.6) values. Note that TV and EV indicate the true (actual) and estimated values respectively.

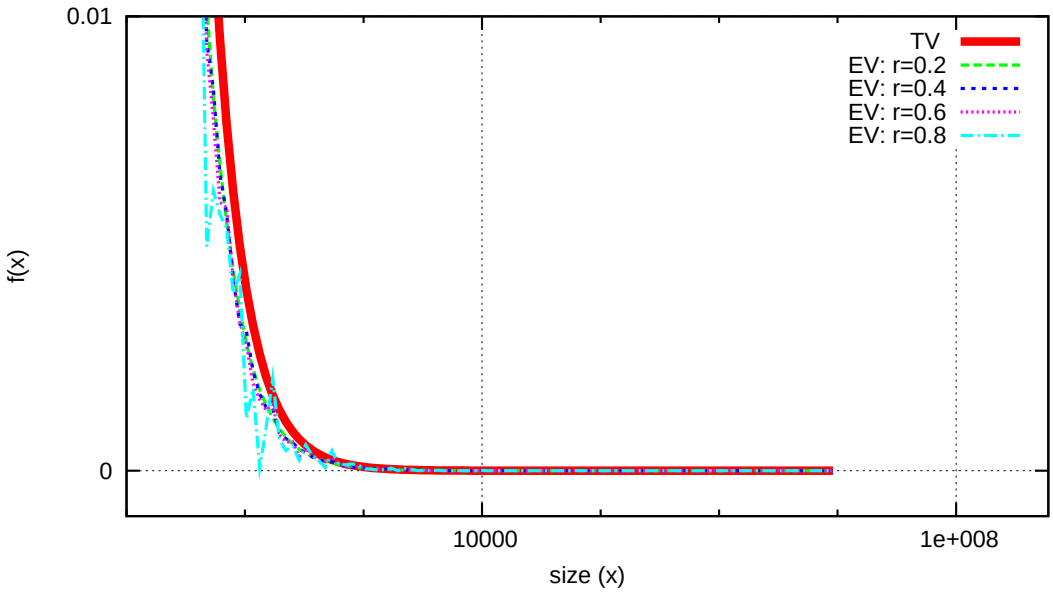
Let us first discuss how well the kernel density estimator performs when it comes to estimating Bounded Pareto distributions. In Figures 6.5, 6.6 and 6.7 we have considered three different Bounded Pareto distributions with three different α values, namely, 0.7, 1.4 and 2.1. These α values have been chosen to cover a wide range traffic scenarios [Crovella and Bestavros, 1997; Crovella et al., 1998b]. As far as probability density estimation (PDF) for Bounded Pareto distribution is concerned, we note that there is a good overall fit for almost all r values considered. For most of r values considered, we cannot distinguish between the actual PDF and the estimated PDF due to the differences between the estimated values and actual values being very small. We note that when $r = 0.8$, the estimated PDF is not smooth, however, it still imitates the actual distribution.

When it comes to estimating the cumulative distribution function (CDF), we note there are minor variations in the estimated CDF depending on r values used. For example, when $r = 0.2$, the estimated CDF lies slightly below the actual CDF, while, when $r = 0.8$ the estimated CDF lies slightly above the actual distribution.[9] We also note that there are cases, where the estimated PDF is not a very smooth function but corresponding CDF is a smooth function imitating the actual cumulative distribution function. The reason for this is that when we estimate CDF (using the estimated PDF), the roughness of the estimated PDF disappears because the estimation of CDF involves numerical integration.

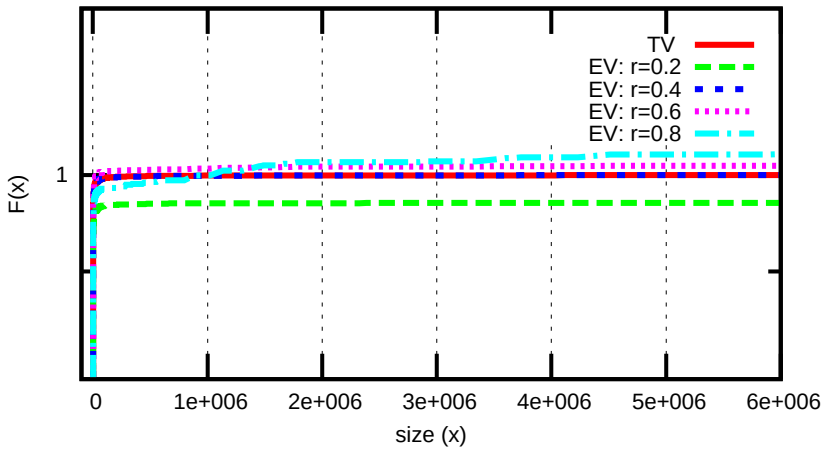
Let us now investigate results obtained for Bounded exponential distribution. These results are shown in Figures 6.8, 6.9 and 6.10. As far the estimation of PDF is concerned, the estimated PDF imitates the actual PDF under a wide range of r values. For a few cases, however, the estimated PDF significantly deviates from the actual PDF. For example, when $\lambda = 0.01$ and $r = 0.2$, we note that the estimated PDF not imitating the actual PDF. When it comes to estimating CDF, the results are better compared to the estimation of PDF.

Finally, let us now consider the results obtained for Bounded Weibull distribution (refer to Figures 6.11, 6.12 and 6.13). We perform the density estimation for three different Bounded Weibull distributions with three different α parameters, namely, 0.5, 1.5 and 2.5. We have chosen these

[9] Note that the y axis does not start from 0 rather it starts from value closer to 1. This has been done in order to clearly illustrate the changes that occur when y is close to 1.



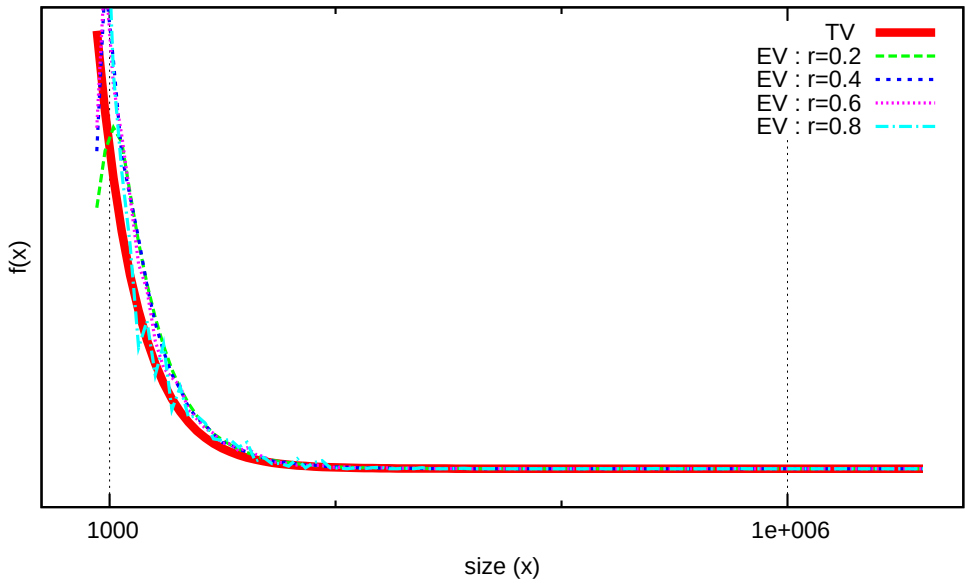
Estimation of PDF



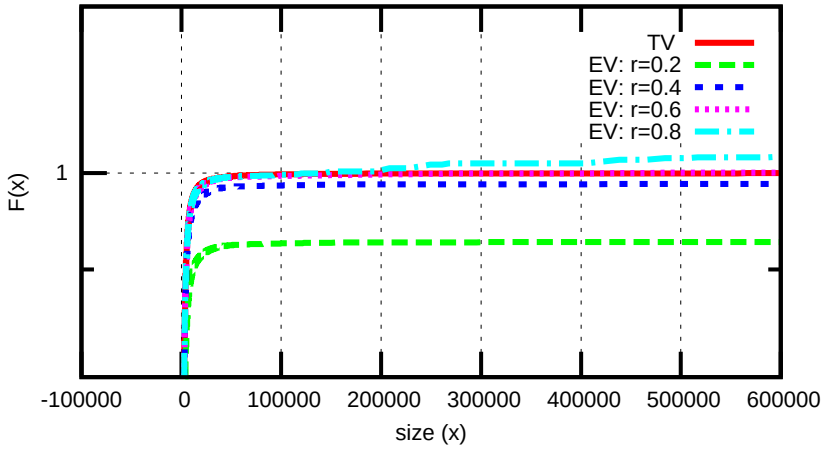
Estimation of CDF

Figure 6.5: Estimating Bounded Pareto distribution: $\alpha = 0.5$

parameters so that we get Bounded Weibull distributions with different properties particularly with different variances and shapes as illustrated in Figures 6.11, 6.12 and 6.13. Note that the estimated PDF for Weibull distribution imitates the actual PDF except when $r = 0.8$, where the estimated PDF has a rough curve. On the other hand, the estimated CDF imitates the actual CDF under all r values.



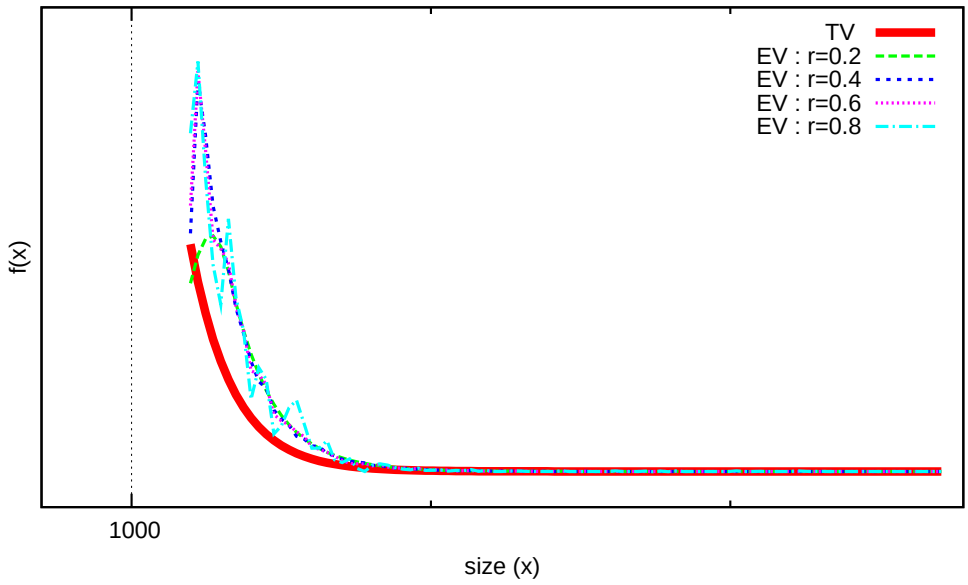
Estimation of PDF



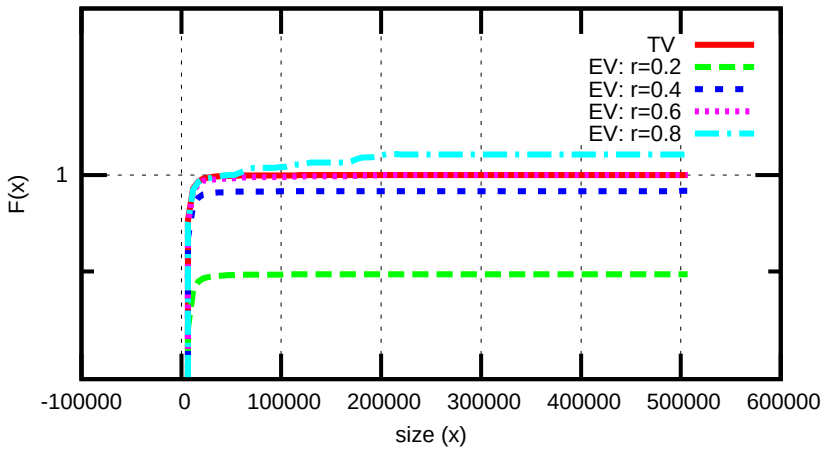
Estimation of CDF

Figure 6.6: Estimating Bounded Pareto distribution: $\alpha = 1.4$

We note that (non-parametric) kernel density estimators perform well in estimating different types of distributions. Moreover, we note that r has a significant effect on the quality of the fit; hence, the accuracy of results. Most of the r values, produced very good overall fit, in particular when estimating the cumulative distribution function. However, the question is which r value will result in the best



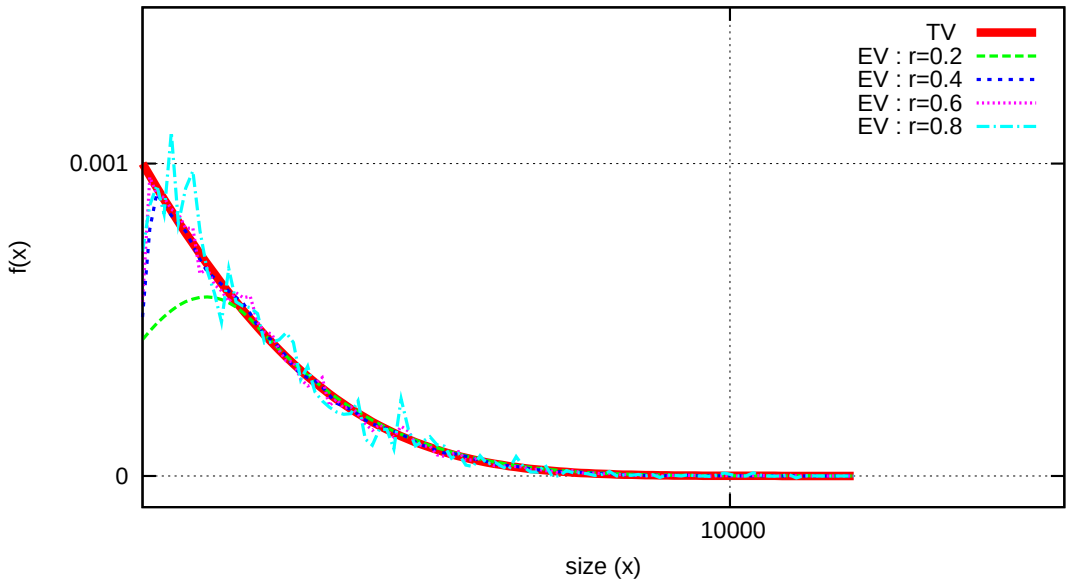
Estimation of PDF



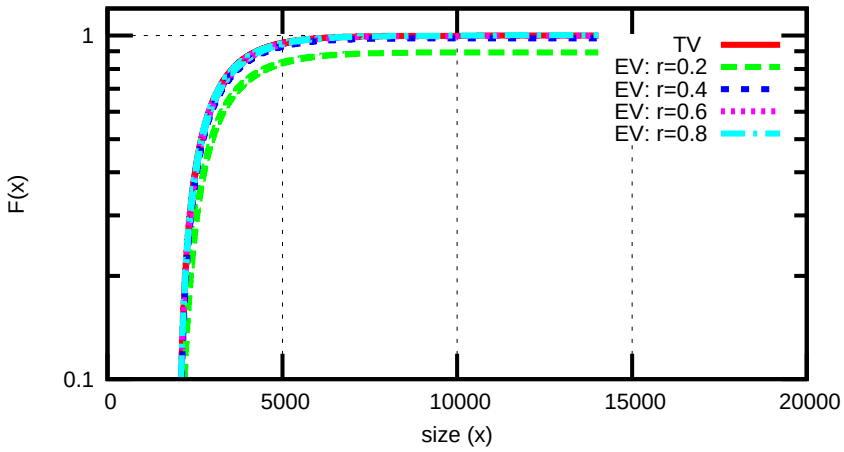
Estimation of CDF

Figure 6.7: Estimating Bounded Pareto distribution: $\alpha = 2.1$

fit. It was clear from our experiments that the r value that results in the most accurate fit varies from distribution to distribution and varies for different parameters for the same distribution. As such, we compute the expected waiting time of policies using different r values and then take the average performance. It important to note that estimation of the probability density function and cumulative



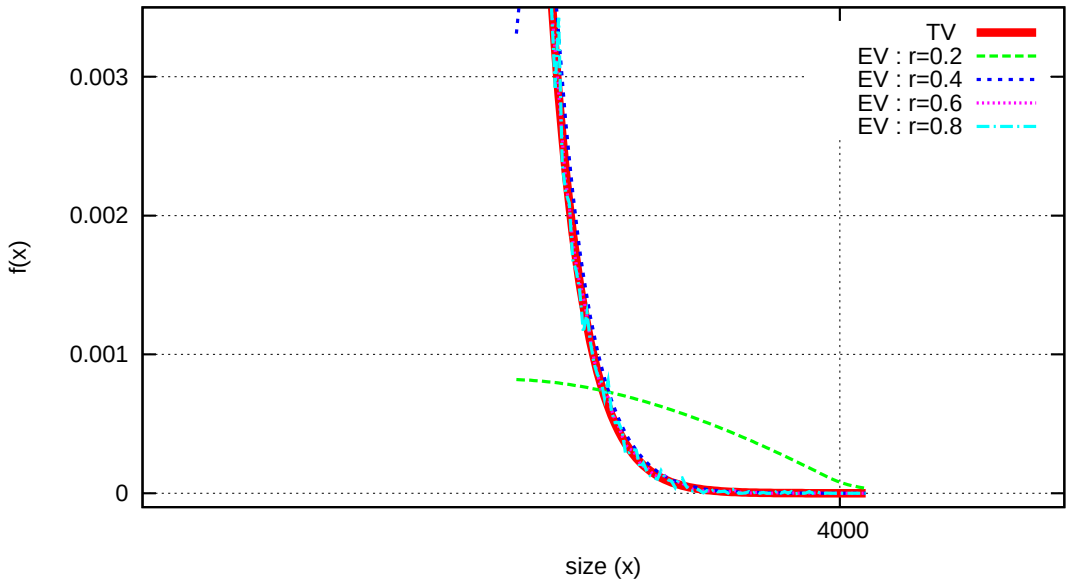
Estimation of PDF



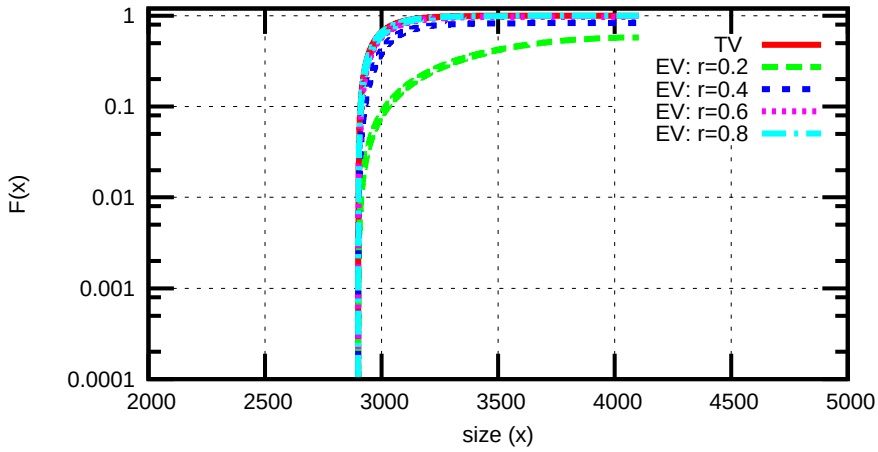
Estimation of CDF

Figure 6.8: Estimating Bounded exponential distributions: $\lambda = 0.001$

distribution function multiple times can be computationally intensive. However, the frequency at which these computations are carried out is low because ADAPT-POLICY only performs the density estimation after it collects n number of service times, where $n > 20000$ (refer to Section 6.3). For example, in the simulation results presented in the following section, even though there 50000 tasks



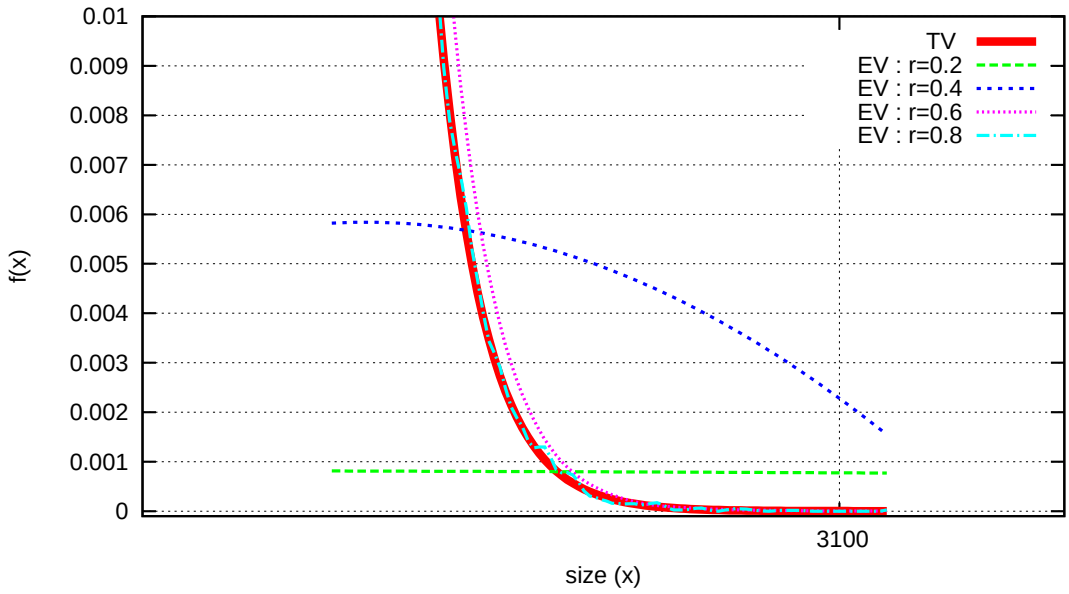
Estimation of PDF



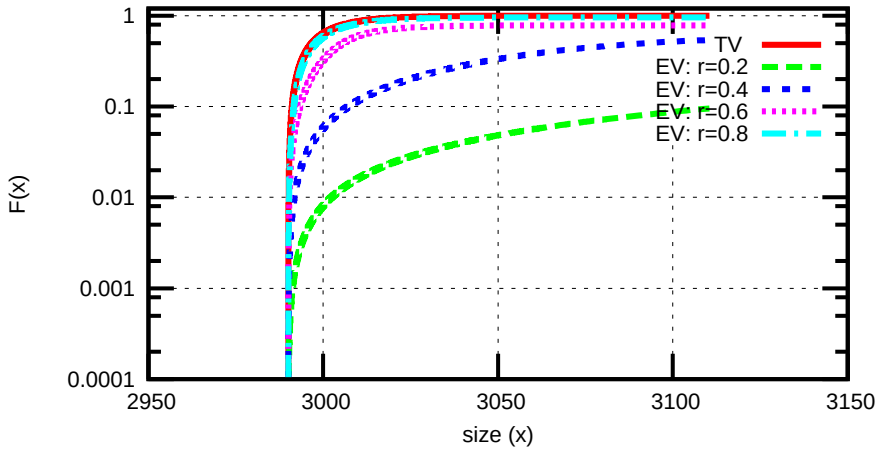
Estimation of CDF

Figure 6.9: Estimating Bounded exponential distributions: $\lambda = 0.01$

in total, the density estimation is only being performed a maximum of 20 times.



Estimation of PDF

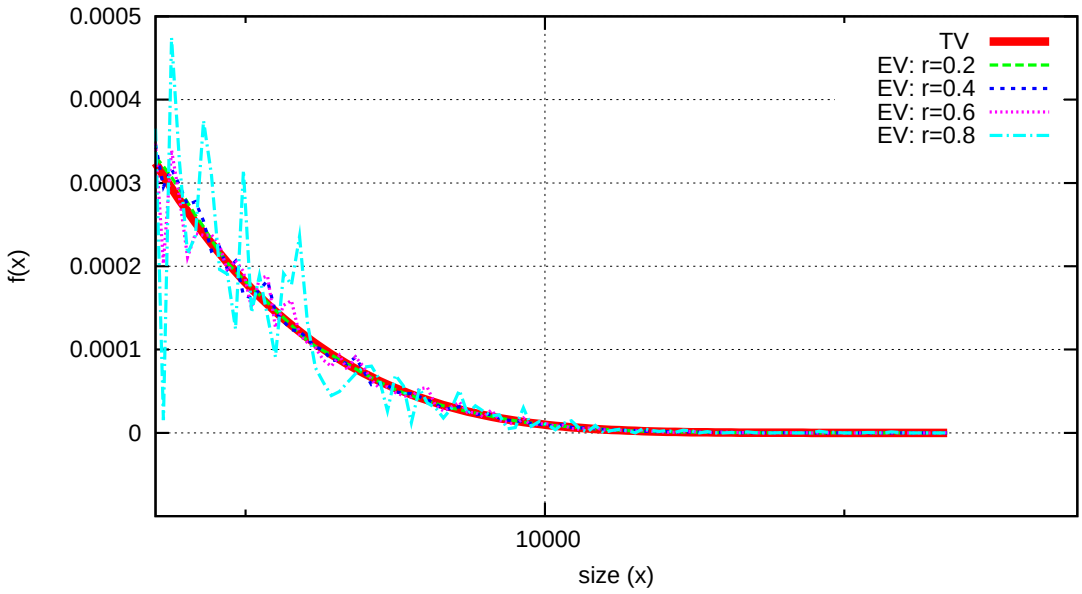


Estimation of CDF

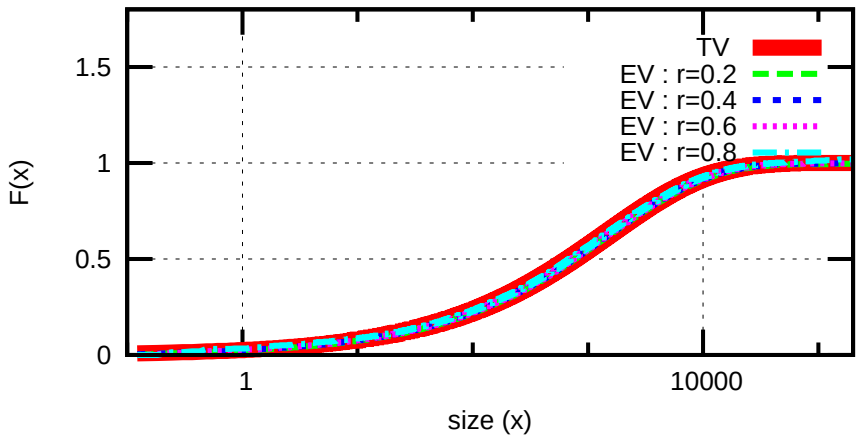
Figure 6.10: Estimating Bounded exponential distributions: $\lambda = 0.1$

6.3 Experimental analysis

Previous section showed that non-parametric density estimators can be successfully applied for estimating different types of distributions with different properties. This section provides simulation



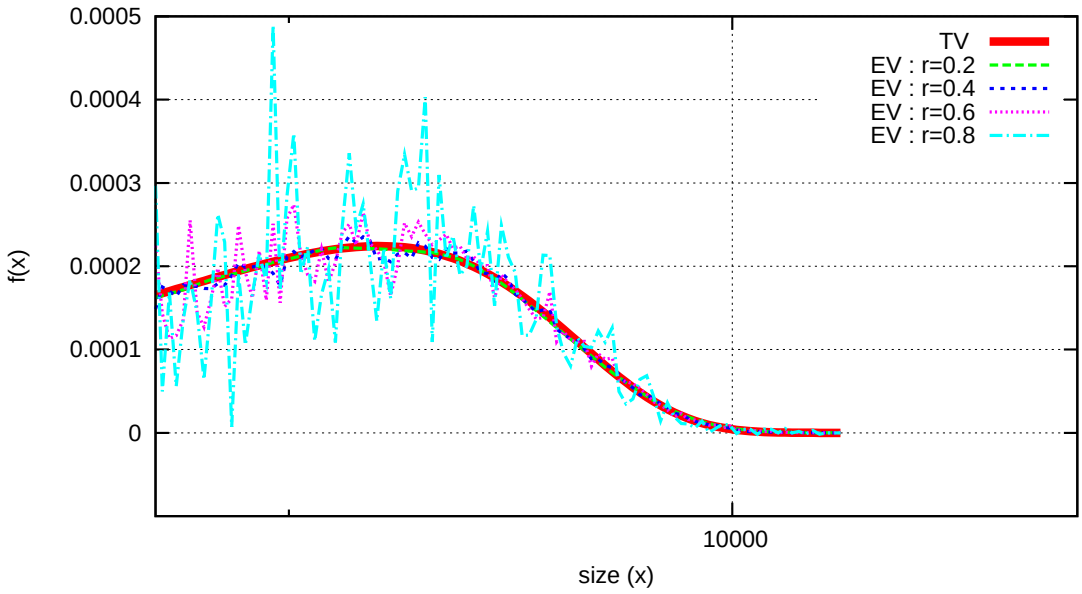
Estimation of PDF



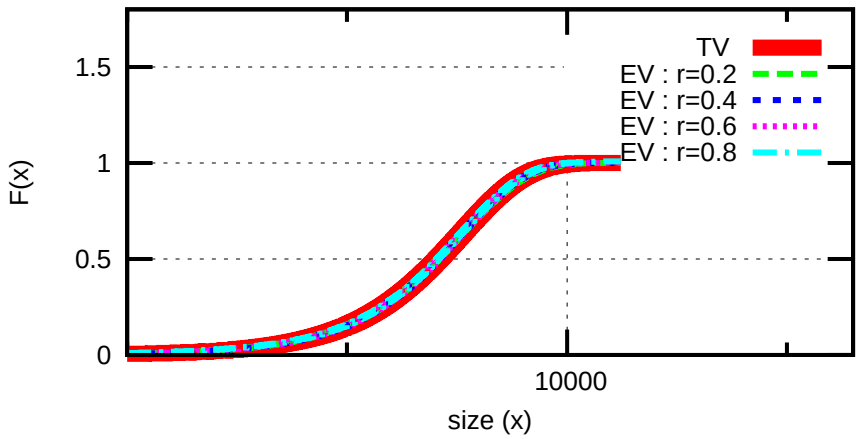
Estimation of CDF

Figure 6.11: Estimating Bounded Weibull distributions: $\alpha = 0.5$

results for ADAPT-POLICY, which is based on the non-parametric techniques discussed in the previous section. We consider task assignment in a server farm that does not support preemptive migration. As discussed, there exist a number of core policies, which can be utilised in such a system. Since the aim of this section is to demonstrate the main concepts behind ADAPT-POLICY, we limit the number



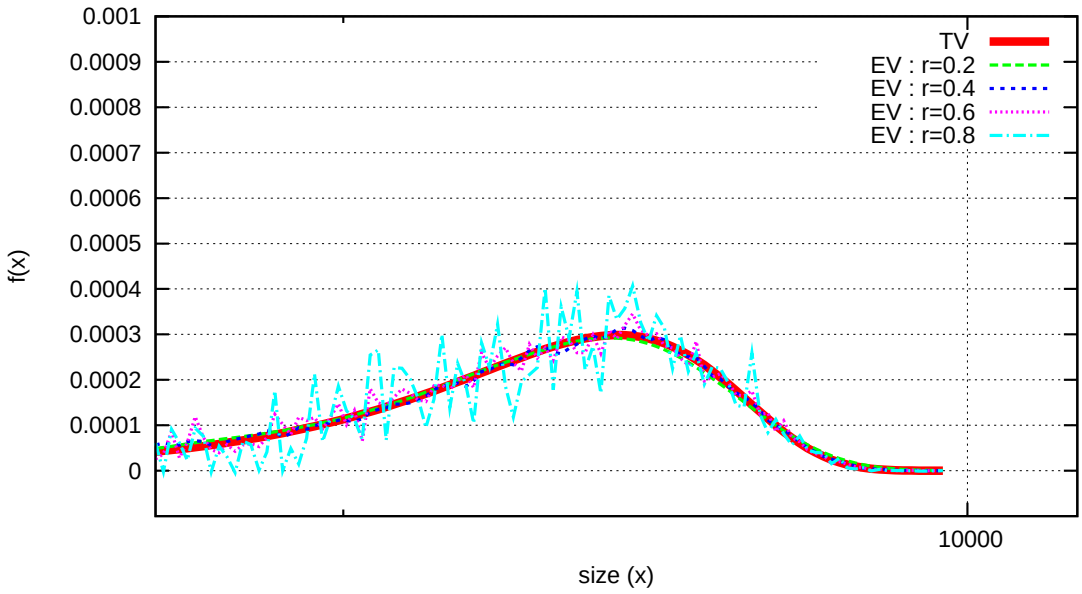
Estimation of PDF



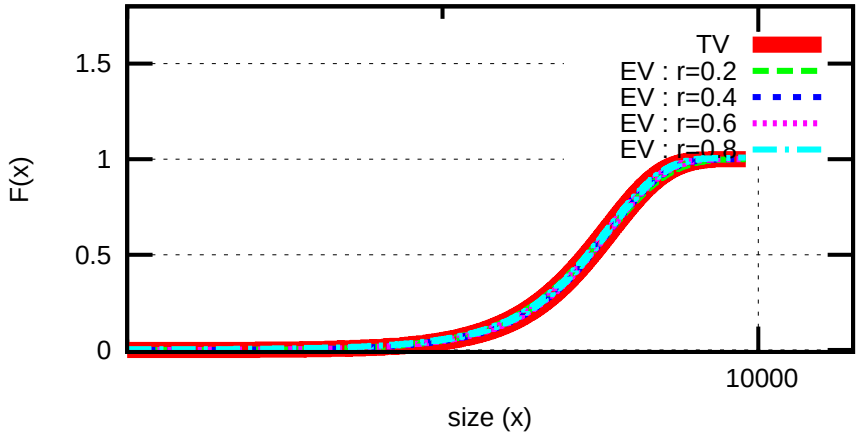
Estimation of CDF

Figure 6.12: Estimating Bounded Weibull distributions: $\alpha = 1.5$

of task assignment policies in the policy list (of ADAPT-POLICY) to two and these are TAGS and Random. We compare the performance of ADAPT-POLICY with STATIC-TAGS, ADAPT-TAGS. Details of these policies are given below. Note that the criteria for generating test data for simulations are discussed in Section 6.3.2.



Estimation of PDF



Estimation of CDF

Figure 6.13: Estimating Bounded Weibull distributions: $\alpha = 1.5$

- Random: Refer to Section 2.3.
- STATIC-TAGS: TAGS [Harchol-Balter, 2002] is one of the most popular task assignment policies that has shown better performance under a range of workload conditions in particular highly variable traffic conditions. This STATIC-TAGS is identical to TAGS. The term STATIC

has been used simply to indicate that TAGS has fixed cut-offs. In the simulations we use two variants of STATIC-TAGS.

  - STATIC-TAGS-HIGH-VAR: The cut-offs for STATIC-TAGS-HIGH-VAR are computed to optimise the performance under very high task size variances.
  - STATIC-TAGS-LOW-VAR: The cut-offs for STATIC-TAGS-LOW-VAR are computed to optimise the performance under low task size variances.

- ADAPT-TAGS: To be more compatible with ADAPT-POLICY, we introduce a new version of TAGS called ADAPT-TAGS. ADAPT-TAGS is based on the non-parametric density estimation techniques that we have introduced in Section 6.1 for ADAPT-POLICY. The difference between ADAPT-TAGS and ADAPT-POLICY is that ADAPT-POLICY has the ability to adaptively change the assignment policy and the cut-offs, whereas ADAPT-TAGS can only change the cut-offs of TAGS on-line.

6.3.1 Simulation algorithm

The simulation model, as illustrated in Figure 6.14, consists of a task generator, a switch, a number of back-end hosts and a sink device. The task generator generates tasks using different service time distributions and these tasks are sent to the switch (central dispatcher), which distributes these tasks among the back-end hosts according to a specific task assignment policy. The calculations for ADAPT-POLICY are implemented at the sink device and it is configured in such a way that each time a host completes processing a task, a message is sent to the sink device. The following is the algorithm executed at the sink, whenever a new message arrives at the sink.

$sampling\_time = 0.0$ /*set once when the simulation starts up*/
$task\_count = 0$ /*set once when the simulation starts up*/
if $current\_time() \geq next\_sampling\_time$ **then**
  if $task\_count = 0$ *AND* $task.arrival\_time() \leq current\_time()$ **then**
    $lower\_id = task.get\_my\_id()$
    $upper\_ld = task.get\_my\_id() + window\_size$
  end if
  if $lower\_id \leq task.get\_my\_id()$ *AND* $upper\_id \geq task.get\_my\_id()$ **then**
    $task\_count = task\_count + 1$
    $task\_size\_vec.push\_back(task.get\_my\_size())$
  end if
  if $task\_count = window\_size$ **then**
    $task\_count = 0, lower\_id = 0, upper\_id$

```
        next_sampling_time = sampling_interval + next_sampling_time
        for r = 0.9 r ≥ 0.2 r = r − 0.1 do
            estimate_pdf(task_size_vec)
            estimate_cdf(task_size_vec)
            estimate_m1(task_size_vec)
            estimate_m2(task_size_vec)
            if cdf_valid then
                for eachpolicyinpolicy_list do
                    e_w[r][policy_id] = compute_e_w(policy_id, m1, m2, pdf, cdf, pso)
                end for
            end if
        end for
        find_best_policy(performance[r][policy_id], criteria)
        deploy_new_policy()
    end if
    next_sampling_time = next_sampling_time + sampling_time
    task_count = 0
end if
```

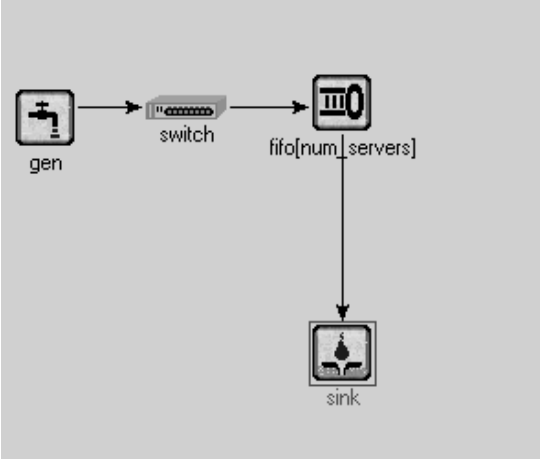


Figure 6.14: OMNET++ simulation model for ADAPT-POLICY

Let us now discuss the above algorithm in more detail. The *next_sampling_time* parameter of the above algorithm allows the ADAPT-POLICY to be configured in two different operating modes, namely, the exhaustive mode and non-exhaustive mode. To perform the density estimation ex-

haustively, *next_sampling_time* is set at 0 and to perform the density estimation non-exhaustively *next_sampling_time* is set at a higher value (e.g. 10 hours).[10] Figures 6.15 and 6.16 illustrate the timing diagram for exhaustive and non-exhaustive modes respectively. The exhaustive density estimation will be useful for rapidly changing service time distributions. If the service time distribution of tasks do not vary at a higher rate then the density estimation can be performed at a lower frequency (i.e. non-exhaustive density estimation) by increasing the vale of *sampling_time* to a higher value.

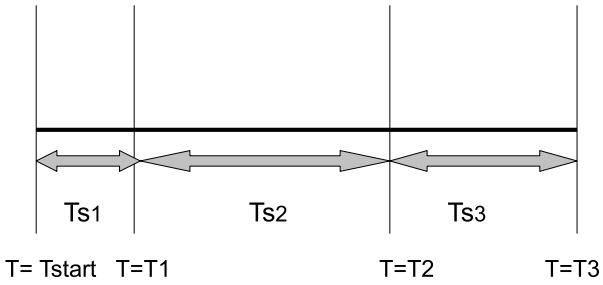


Figure 6.15: Timing plot for ADAPT-POLICY under exhaustive mode

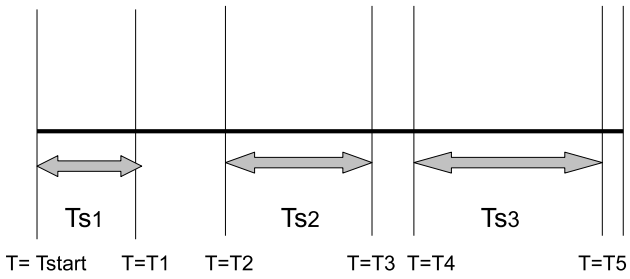


Figure 6.16: Timing plot for ADAPT-POLICY under non-exhaustive mode

We note that when ADAPT-POLICY operates on exhaustive mode it collects the first batch of service

[10]Each time a new message arrives at the sink, it checks whether *next_sampling_time* is greater than or equal to the current system time. If this condition is true, this means that the sink can start storing the service times of tasks that complete their execution in the system.

times during the period $T = T_{start}$ and $T = T_1$, the second batch of service times during the period $T = T_1$ and $T = T_2$ and so on. During each period it collects *window_size* number of service times, where *window_size* is set at the fixed value of 20000.[11] Immediately after collecting *window_size* number of service times, ADAPT-POLICY estimates the service time distribution and its distributional properties (multiple times) using different r values by applying the techniques presented in Section 6.1.2.[12] This calculation results in multiple sets of PDF, CDF and moment arrays, where each set corresponds to a different r value. For each r and each task assignment policy, ADAPT-POLICY then computes the expected waiting time. This calculation results in multiple expected waiting times for each policy. ADAPT-POLICY computes the final expected waiting time for each policy by taking the average of these individual expected waiting times. The task assignment policy with the least final expected waiting time is then used for assigning the next batch of tasks. The time it takes to perform these computations are not shown in the timing diagram because it is not significant compared to the time it takes to collect the sample, which will vary depending on the arrival rates of tasks into the system. In fact, the total computational time depends on numerous factors such as the sample size, the number of times density estimation is performed (using different r) and the configuration of the PSO algorithm (if it is used). Immediately after ADAPT-POLICY finishes these computations, it starts collecting the data for the next sample and so on.

6.3.2 Experimental results

This section presents the simulation results for the policies discussed in Section 6.3. The simulation model is developed using the C++ based OMNET++ discrete event simulator. Some important details about the simulations are given below.

- The task generator of the simulator is configured to generate tasks from three different service time distributions, namely, the Bounded exponential, Bounded Pareto and Bounded Weibull. The parameters for these distributions have been selected to cover a wide range of traffic patterns.[13] Which distribution to generate traffic from and which parameters to use for the chosen distribution are chosen randomly, i.e. from (different) uniform distributions.[14]

[11] The reason for using 20000 as the sample size is discussed in Section 6.3.2

[12] Note that r determines the bandwidth of the kernel estimator (refer to Equation 6.6).

[13] It is assumed that the services times of tasks have a fixed upper bound of 10^7 and a mean of 3000 [Harchol-Balter et al., 1999]. The reason for using these values has been discussed in Section 2.1.2.

[14] For Bounded Pareto and Bounded Weibull distributions α parameter is generated randomly, whereas for the exponential distribution λ is generated randomly. Once these parameters are determined the lower bound for the distribution is computed such that the upper bound is 10^7 and the mean is 3000 (refer to Section 2.1.2).

- Once the distribution is selected, the task generator generates n number of tasks from the chosen distribution, where n is a uniform random variable, which determines the rate at which the distribution changes. We define three different distribution change rates.
  - Moderate: $n \sim U(75000.0, 85000.0)$
  - High: $n \sim U(35000.0, 45000.0)$
  - Very high: $n \sim U(15000.0, 25000.0)$
- The tasks that are generated by the task generator are sent to the central dispatcher of the server farm, which distributes these tasks among the back-end hosts according to one of the task assignment policies discussed in Section 6.3. The system load is maintained at 0.5 for all the simulations by controlling the average arrival rate.
- The moving average of waiting time is used as the performance metric. Simulation results are provided under the three distribution change rates (i.e. moderate, high and very high) discussed above. For each distribution, change rate two sets of simulation results are provided using two different seeds generated using the seed generating tools available in OMNET++. Simulation results under seed 1 and 2 are called the Run 1 and Run 2 respectively.
- The sample size for density estimation is set at the fixed value 20000. This value has been selected after conducting numerous experiments using different service time distributions. In these experiments, it was noted that sample size must be at least 20000 in order to accurately estimate the service time distribution of tasks.[15] For certain distributions (e.g. once with the low variance and low skewness), the sample size does not have to be as high as 20000. Large sample sizes are only required for particular types of distributions (e.g. Pareto distributions with low α value).

(a) Moderate distribution change rates

Let us first investigate the performance of policies under moderate distribution change rates. Figures 6.17 and 6.18 plot the moving average (of waiting time for policies) for Run 1 and Run 2 respectively, while Tables 6.1 and 6.2 present the distribution change profile for ADAPT-POLICY under Run 1 and Run 2 respectively.

These tables are produced using the log files, which are created by the simulator. In Tables 6.1 and 6.2, POLICY ID denotes the ID of the current policy that ADAPT-POLICY uses to assign tasks.

[15] Note that previous simulations that investigate the performance of policies have used similar sample sizes [Broberg et al., 2006; 2004].

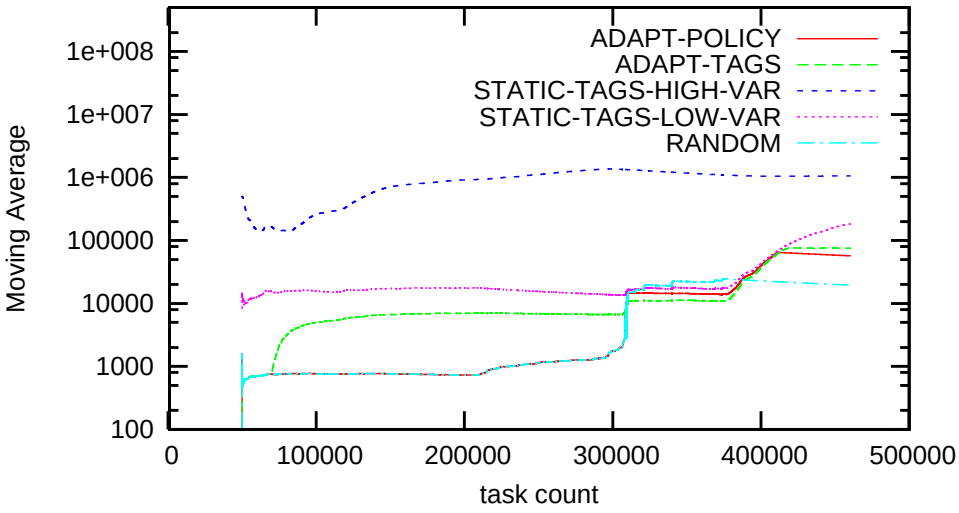


Figure 6.17: Moving average for policies under moderate distribution change rates: Run 1

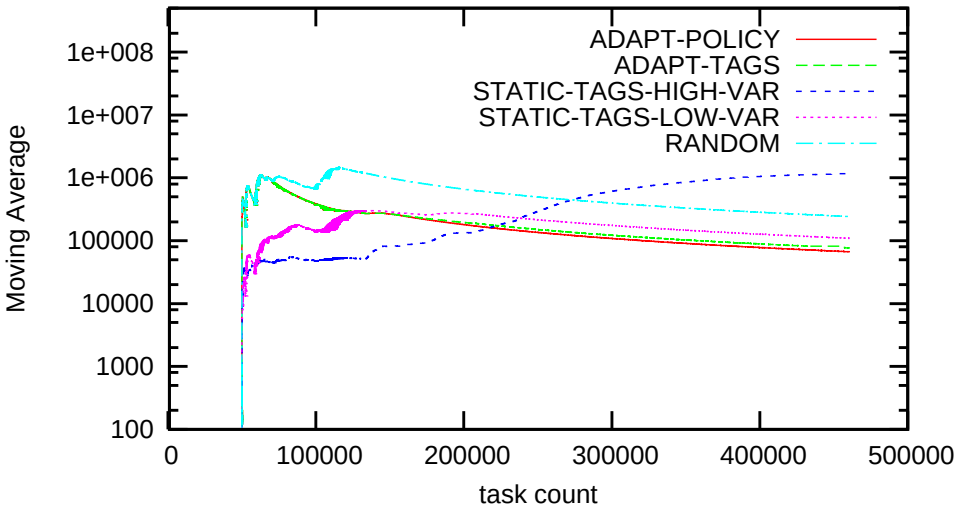


Figure 6.18: Moving average for policies under moderate distribution change rates: Run 2

As pointed out, we provide simulation results for ADAPT-POLICY, when the task assignment policy list of ADAPT-POLICY consists of the two task assignment policies; Random and TAGS. In Tables 6.1 and 6.2, POLICY ID = 0 and POLICY ID = 1 denote Random and TAGS respectively. In addition

Table 6.1: ADAPT-POLICY: Run 1*, distribution change rate: moderate*

1	distribution = boundedweibull: alpha = 2.5, p = 1e+07, beta = 3381, task_id = 50000
2	POLICY ID = 0, task_id = 70003
3	POLICY ID = 0, task_id = 90006
4	POLICY ID = 0, task_id = 110010
5	POLICY ID = 0, task_id = 130011
6	distribution = boundedexponential: k = 2000, p = 1e+07, rate = 0.001, task_id = 131697
7	POLICY ID = 0, task_id = 150015
8	POLICY ID = 0, task_id = 170016
9	POLICY ID = 0, task_id = 190017
10	distribution = boundedpareto: k = 1571.53, p = 1e+07, alpha = 2.1, task_id = 207440
11	POLICY ID = 0, task_id = 210019
12	POLICY ID = 0, task_id = 230021
13	POLICY ID = 0, task_id = 250021
14	POLICY ID = 0, task_id = 270025
15	POLICY ID = 0, task_id = 290028
16	distribution = boundedpareto: k = 878.08, p = 1e+07, alpha = 1.4, task_id = 291900
17	CHANGE POLICY: POLICY ID = 1 CUT OFFS: 10876.6, 1e+07, task_id = 310572
18	POLICY ID = 1 CUT OFFS: 21408.7, 1e+07, task_id = 330579
19	POLICY ID = 1 CUT OFFS: 8746.52, 1e+07, task_id = 350594
20	POLICY ID = 1 CUT OFFS: 8896.83, 1e+07, task_id = 370596
21	distribution = boundedexponential: k = 2900, p = 1e+07, rate = 0.01, task_id =376740
22	POLICY ID = 1 CUT OFFS: 3777.36,1e+07, task_id = 390683
23	CHANGE POLICY: POLICY ID = 0, task_id = 411024
24	POLICY ID = 0, task_id = 431028
25	POLICY ID = 0, task_id = 451033

to POLICY ID, the tables also show the current task id and the optimal scheduling parameters for the current task assignment policy if it has any scheduling parameters. The term CHANGE POLICY is used in the tables to indicate that there has been a change in the task assignment policy.

Let us discuss the moving average for policies under Run 1. Note from Table 6.1 that in the first half of Run 1, the service times have been generated using the service distributions with relatively low variances. Therefore, STATIC-TAGS-LOW-VAR performs well in this run as its scheduling parameters have been computed to optimise the performance under low task size variances. On the other hand, STATIC-TAGS-HIGH-VAR has the worst performance because scheduling parameters for STATIC-TAGS-HIGH-VAR have been computed to optimise the performance under high task size variabilities. ADAPT-TAGS outperforms both STATIC-TAGS-LOW-VAR and STATIC-TAGS-HIGH-VAR because unlike STATIC-TAGS-HIGH-VAR and STATIC-TAGS-LOW-VAR, ADAPT-TAGS can dynamically compute its optimal parameters according to changes that occur in the incoming service time distribution.

Both ADAPT-POLICY and Random perform extremely well. Note that ADAPT-POLICY outperforms Random, when *task_id* is in the range 300000 and 400000, whereas Random outperforms

Table 6.2: ADAPT-POLICY: Run 2, distribution change rate: moderate

1	distribution = boundedpareto: k = 28.5212, p = 1e+07, alpha = 0.7
2	POLICY ID = 1 CUT OFFS: 96679.6, 1e+07, task_id = 70004
3	POLICY ID = 1 CUT OFFS: 117927, 1e+07, task_id = 90022
4	POLICY ID = 1 CUT OFFS: 534884, 1e+07, task_id = 114650
5	distribution = boundedexponential: k = 2900, p = 1e+07, rate = 0.01
6	POLICY ID = 1 CUT OFFS: 45043.9, 1e+07, task_id = 134755
7	CHANGE POLICY: POLICY ID = 0, task_id = 154769
8	POLICY ID = 0, task_id = 174774
9	POLICY ID = 0, task_id = 194777
10	distribution = boundedweibull: alpha = 2.5, p = 1e+07, beta = 3381
11	POLICY ID = 0, task_id = 214778
12	POLICY ID = 0, task_id = 234779
13	POLICY ID = 0, task_id = 254781
14	POLICY ID = 0, task_id = 274783
15	distribution = boundedweibull: alpha = 1.5, p = 1e+07, beta = 3323.2
16	POLICY ID = 0 CUT OFFS, task_id = 294786
17	POLICY ID = 0 CUT OFFS, task_id = 314792
18	POLICY ID = 0 CUT OFFS, task_id = 334793
19	POLICY ID = 0 CUT OFFS, task_id = 354794
20	POLICY ID = 0 CUT OFFS, task_id = 374797
21	distribution = boundedpareto: k = 1571.53, p = 1e+07, alpha = 2.1
22	POLICY ID = 0, task_id = 394797
23	POLICY ID = 0, task_id = 414800
24	POLICY ID = 0, task_id = 434801
25	POLICY ID = 0, task_id = 454802

ADAPT-POLICY when *task_id* is in the range 400000 and 450000. Note from Table 6.1 that there is a change in the service time distribution from Bounded Pareto with α = 2.1 to Bounded Pareto with α = 1.4 at $task\_id = 291900$. The optimal policy for Bounded Pareto with α = 1.4 is TAGS, which is initiated at 310572 (after performing the density estimation).

Let us now briefly discuss the moving average for policies for Run 2 under moderate distribution change rates. We note from Table 6.2 that in the initial stages of Run 2, the moving average (of waiting time) for Random increases rapidly up to *task_id* 114650. This is because Random has very poor performance under a Bounded Pareto distribution with low α values, which has been used to generate traffic at the start. Also note that STATIC-TAGS-HIGH-VAR performs well until $task\_id = 114650$ as it is the optimal policy for assigning tasks until $task\_id = 114650$. However, from there onwards its performance degrades significantly as the traffic becomes less variable later on. ADAPT-POLICY performs well for both Run 1 and Run 2 since it has the ability to adaptively change the task assignment policy and the scheduling parameters based on the most recent traffic conditions.

(b) High distribution change rates

Figure 6.19 and Table 6.3 illustrate the moving average and the distribution profile for Run 1 respectively under high distribution change rates. First we note that the moving average for Random

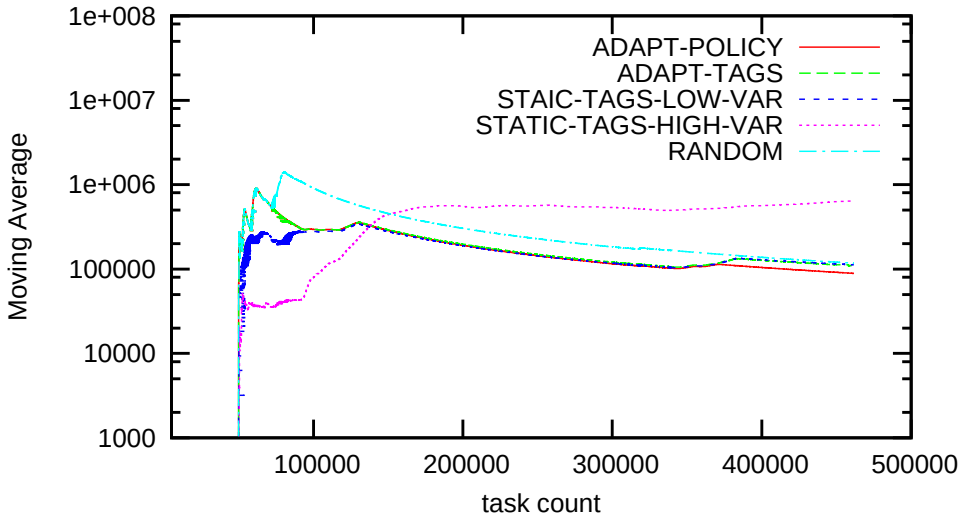


Figure 6.19: Moving average for policies under high distribution change rates: Run 1

increases rapidly at the start of Run 1 and then the moving average decreases. At the start of the Run 1, Bounded Pareto distribution with $\alpha = 0.7$ has been used to generate tasks under which the Random has very poor performance. On the other hand, STATIC-TAGS-HIGH-VAR has the lowest moving (the best) average at the start because its scheduling parameters have been computed to optimise the waiting time under service time distributions with high variances (e.g. Bounded Pareto distribution with $\alpha = 0.7$). The moving average of STATIC-TAGS-HIGH-VAR, however, degrades significantly in the later stages because STATIC-TAGS-HIGH-VAR is not the optimal policy for assigning tasks under the rest of service time distributions used in this run.

ADAPT-TAGS, ADAPT-POLICY, STATIC-TAGS-LOW-VAR have similar behaviour for this particular run. However, towards the end of the run ADAPT-POLICY outperforms both STATIC-TAGS-LOW-VAR and ADAPT-TAGS because neither STATIC-TAGS-LOW-VAR nor ADAPT-TAGS is the optimal policy for assigning tasks towards the end of run. At the start of the experiment, ADAPT-POLICY does not perform as well as STATIC-TAGS-HIGH-VAR because it takes some time for it to change the task assignment policy and the scheduling parameters to match with the incoming service time distribution.

Table 6.3: ADAPT-POLICY: Run 1*, distribution change rate: high*

1	distribution = boundedpareto: k = 28.5212, p = 1e+07, alpha = 0.7, task_id = 50000
2	POLICY ID = 1 CUT OFFS: 85313.1, 1e+07, task_id = 70003
3	POLICY ID = 1 CUT OFFS: 115956, 1e+07, task_id = 90016
4	distribution = boundedexponential: k = 2900, p = 1e+07, rate = 0.01, task_id = 91697
5	POLICY ID = 1 CUT OFFS: 3866.2, 1e+07, task_id = 110123
6	distribution = boundedweibull: alpha = 2.5, p = 1e+07, beta = 3381, task_id = 127440
7	CHANGE POLICY: POLICY ID = 0, task_id = 130309
8	POLICY ID = 0, task_id = 150313
9	POLICY ID = 0, task_id = 170318
10	distribution = boundedweibull: alpha = 1.5, p = 1e+07, beta = 3323.2, task_id = 171900
11	POLICY ID = 0, task_id = 190320
12	POLICY ID = 0, task_id = 210323
13	distribution = boundedpareto: k = 1571.53, p = 1e+07, alpha = 2.1, task_id = 216740
14	POLICY ID = 0, task_id = 230324
15	POLICY ID = 0, task_id = 250328
16	distribution = boundedweibull: alpha = 2.5, p = 1e+07, beta = 3381, task_id = 260578
17	POLICY ID = 0, task_id = 270330
18	POLICY ID = 0, task_id = 290333
19	distribution = boundedpareto: k = 878.08, p = 1e+07, alpha = 1.4, task_id = 298945
20	POLICY ID = 1 CUT OFFS: 4605.11, 1e+07, task_id = 310334
21	POLICY ID = 1 CUT OFFS: 8367.33, 1e+07, task_id = 330341
22	distribution = boundedexponential: k = 2990, p = 1e+07, rate = 0.1, task_id = 339933
23	POLICY ID = 1 CUT OFFS: 3471.87, 1e+07, task_id = 350454
24	CHANGE POLICY: POLICY ID = 0, task_id = 370557
25	distribution = boundedweibull: alpha = 2.5, p = 1e+07, beta = 3381, task_id = 381410
26	POLICY ID = 0, task_id = 390559
27	POLICY ID = 0, task_id = 410562
28	distribution = boundedexponential: k = 2000, p = 1e+07, rate = 0.001, task_id = 421626
29	POLICY ID = 0, task_id = 430564
30	POLICY ID = 0, task_id = 450565

The moving average for policies for Run 2 under high distribution rates is illustrated in Figure 6.20. As was done before, the behaviour of moving average can be explained using the distribution change profile illustrated in Table 6.4.

(c) Very high distribution change rates

Let us now consider the moving average of policies under very high distribution change rates. Although one may think ADAPT-POLICY would perform poorly under very high distribution change rates (as it may not be able to cope up with the rate at which the distribution changes), experimental results indicate that this is not the case for most of the scenarios.

Figure 6.21 and Table 6.5 illustrate the moving average and the distribution change profile for Run 1. We note that Random and STATIC-TAGS-HIGH-VAR perform very poorly. On the other hand, STATIC-TAGS-LOW-VAR and ADAPT-TAGS perform well. However, ADAPT-TAGS performs

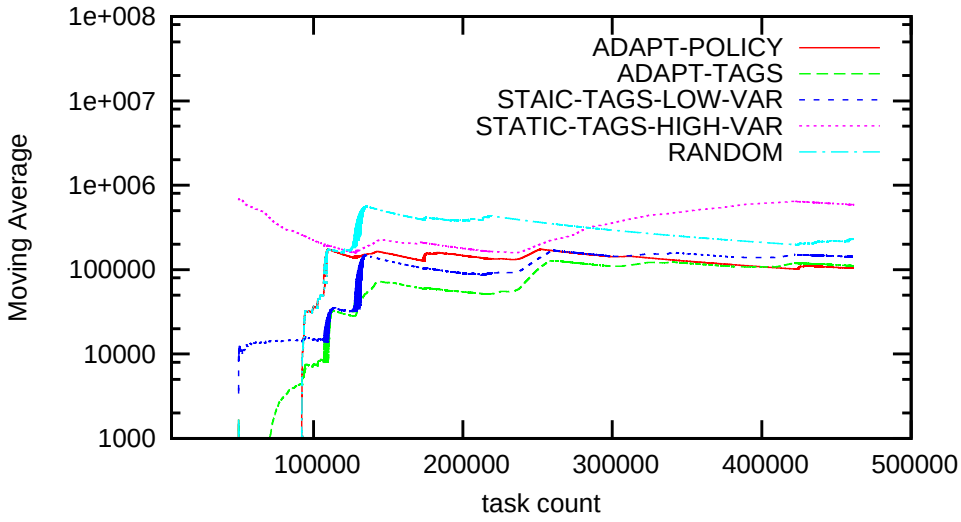


Figure 6.20: Moving average for policies under high distribution change rates: Run 2

better compared to STATIC-TAGS-LOW-VAR because ADAPT-TAGS can dynamically adjust its scheduling parameters according to the changes that occur in the incoming service time distribution of tasks. On the other hand, ADAPT-POLICY, outperforms both ADAPT-TAGS and STATIC-TAGS-LOW-VAR under a wide range of task ids. Figure 6.22 illustrates behaviour of policies for Run 2 under very high distribution change rates. We can explain the moving average for policies using the distribution profile presented in Table 6.6.

(d) Overall expected waiting time

So far, we have considered the moving average of waiting time of tasks under three distribution change rates. Let us now discuss the (overall) expected waiting time for policies. Table 6.7 illustrates the policy with the best overall expected waiting time. We note that out of six cases that we considered ADAPT-POLICY outperforms all other policies in four cases. Recall that in this chapter we only considered moderate, high and very high distribution change rates. We expect the performance of ADAPT-POLICY to be much better under low distribution change rates.

6.4 Conclusion

This chapter proposed a novel policy for assigning tasks in server farms. The proposed policy made no assumptions regarding the underlying service time distribution of tasks or the actual sizes of tasks.

Table 6.4: ADAPT-POLICY: Run 2*, distribution change rate: high*

1	distribution boundedweibull: alpha = 2.5, p = 1e+07, beta = 3381, task_id = 91697
2	CHANGE POLICY: POLICY ID = 0, task_id = 70002
3	CHANGE POLICY: POLICY ID = 0, task_id = 90004
4	distribution = boundedpareto: k = 28.5212, p = 1e+07, alpha = 0.7, task_id = 127440
5	CHANGE POLICY: POLICY ID = 1 CUT OFFS: 73025.3, 1e+07, task_id = 110173
6	distribution = boundedpareto: k = 1571.53, p = 1e+07, alpha = 2.1, task_id = 171900
7	CHANGE POLICY: POLICY ID = 1 CUT OFFS: 56609.3, 1e+07, task_id = 131072
8	CHANGE POLICY: POLICY ID = 0, task_id = 151125
9	CHANGE POLICY: POLICY ID = 0, task_id = 171145
10	distribution = boundedpareto: k = 28.5212, p = 1e+07, alpha = 0.7, task_id = 216740
11	CHANGE POLICY: POLICY ID = 1 CUT OFFS: 201103, 1e+07, task_id = 191147
12	CHANGE POLICY: POLICY ID = 1 CUT OFFS: 65161.4, 1e+07, task_id = 211208
13	distribution = boundedexponential: k = 2900, p = 1e+07, rate = 0.01, task_id = 260578
14	CHANGE POLICY: POLICY ID = 1 CUT OFFS: 24602.5, 1e+07, task_id = 231230
15	CHANGE POLICY: POLICY ID = 0, task_id = 251506
16	distribution = boundedweibull: alpha = 0.5, p = 1e+07, beta = 1500, task_id = 298945
17	CHANGE POLICY: POLICY ID = 1 CUT OFFS: 3972.63, 1e+07, task_id = 271510
18	CHANGE POLICY: POLICY ID = 1 CUT OFFS: 7525.57, 1e+07, task_id = 291513
19	distribution = boundedexponential: k = 2900, p = 1e+07, rate = 0.01, task_id = 339933
20	CHANGE POLICY: POLICY ID = 0, task_id = 311576
21	CHANGE POLICY: POLICY ID = 0, task_id = 331582
22	distribution = boundedweibull: alpha = 1.5, p = 1e+07, beta = 3323.2, task_id = 381410
23	CHANGE POLICY: POLICY ID = 0, task_id = 351591
24	CHANGE POLICY: POLICY ID = 0, task_id = 371592
25	distribution = boundedexponential: k = 2990, p = 1e+07, rate = 0.1, task_id = 421626
26	CHANGE POLICY: POLICY ID = 0, task_id = 391594
27	CHANGE POLICY: POLICY ID = 0, task_id = 411597
28	distribution = boundedpareto: k = 28.5212, p = 1e+07, alpha = 0.7, task_id = 461609
29	CHANGE POLICY: POLICY ID = 1 CUT OFFS: 19833.7, 1e+07, task_id = 431598
30	CHANGE POLICY: POLICY ID = 1 CUT OFFS: 43442.4, 1e+07, task_id = 451599

ADAPT-POLICY defines a set of static-based policies for a given distributed system and it then utilises the task assignment policy with the least expected waiting time for assigning the next batch of tasks. The task assignment policy with the least expected waiting time is determined using the service time distribution of tasks and the various properties of the service time distribution that are estimated on-line using the service times of tasks collected over a period of time. The technique ADAPT-POLICY uses to estimate these distributions is called the non-parametric kernel based density estimation. This is a much more general approach to estimation with a wider range of validity than the corresponding parametric method of estimation such as method of moments and maximum likelihood estimation. Through extensive numerical experiments, we showed that ADAPT-POLICY outperforms other policies under a wide range of scenarios.

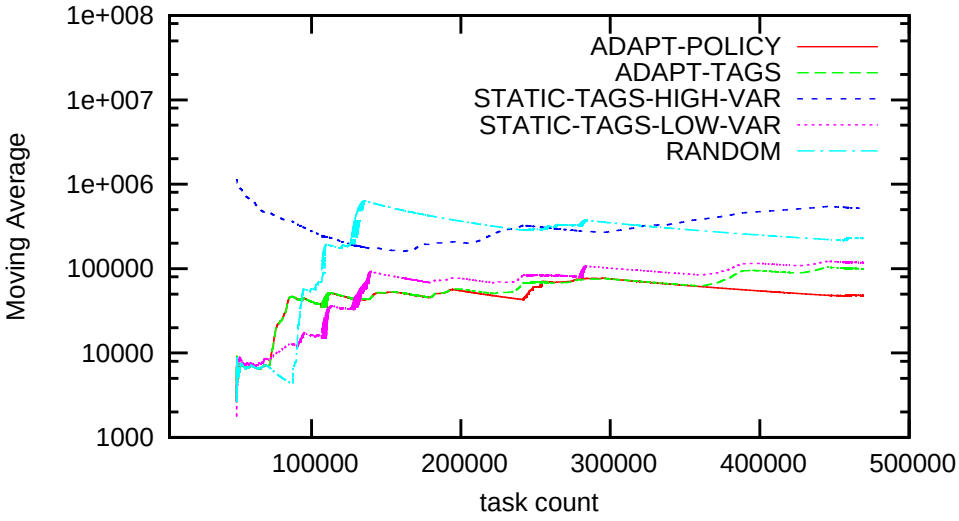


Figure 6.21: Moving average for policies under very high distribution change rates: Run 1

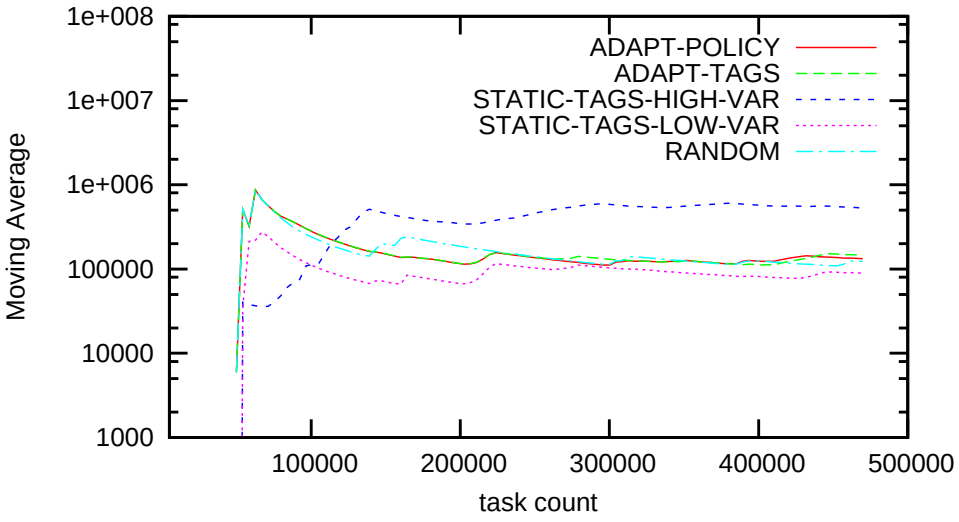


Figure 6.22: Moving average for policies under very high distribution change rates: Run 2

Table 6.5: ADAPT-POLICY: Run 1*, distribution change rate: very high*

1	distribution = boundedweibull: alpha = 0.5, p = 1e+07, beta = 1500, task_id =
2	POLICY ID = 1 CUT OFFS: 7195.84, 1e+07, task_id = 70004
3	distribution = boundedexponential: k = 2000, p = 1e+07, rate = 0.001, task_id = 71697
4	distribution = boundedpareto: k = 28.5212, p = 1e+07, alpha = 0.7, task_id = 87440
5	POLICY ID = 1 CUT OFFS: 4312.76, 1e+07, task_id = 90281
6	POLICY ID = 1 CUT OFFS: 106231, 1e+07, task_id = 111846
7	distribution = boundedpareto: k = 28.5212, p = 1e+07, alpha = 0.7, task_id = 111900
8	POLICY ID = 1 CUT OFFS: 81372.2,1e+07, task_id = 134168
9	distribution = boundedexponential: k = 2000, p = 1e+07, rate = 0.001, task_id = 136740
10	POLICY ID = 1 CUT OFFS: 3968.78, 1e+07, task_id = 154211
11	distribution = boundedweibull: alpha = 0.5, p = 1e+07, beta = 1500, task_id = 160578
12	POLICY: POLICY ID = 1 CUT OFFS: 4780.15, 1e+07, task_id = 174215
13	distribution = boundedexponential: k = 2900, p = 1e+07, rate = 0.01, task_id = 178945
14	CHANGE POLICY ID = 0, task_id = 194251
15	distribution = boundedweibull: alpha = 1.5, p = 1e+07, beta = 3323.2, task_id = 199933
16	POLICY ID = 0, task_id = 214254
17	distribution = boundedexponential: k = 2990, p = 1e+07, rate = 0.1, task_id = 221410
18	POLICY ID = 0, task_id = 234256
19	distribution = boundedpareto: k = 28.5212, p = 1e+07, alpha = 0.7, task_id = 241626
20	CHANGE POLICY: POLICY ID = 1 CUT OFFS: 36300.2, 1e+07, task_id = 254710
21	distribution = boundedpareto: k = 1571.53, p = 1e+07, alpha = 2.1, task_id = 261609
22	POLICY ID = 1 CUT OFFS: 19536.5, 1e+07, task_id = 274817
23	distribution = boundedweibull: alpha = 2.5, p = 1e+07, beta = 3381, task_id = 279111
24	CHANGE POLICY ID = 0, task_id = 294923
25	distribution = boundedexponential: k = 2000, p = 1e+07, rate = 0.001, task_id = 296274
26	POLICY ID = 0, task_id = 314930
27	distribution = boundedpareto: k = 1571.53, p = 1e+07, alpha = 2.1, task_id = 316684
28	POLICY ID = 0, task_id = 334931
29	distribution = boundedexponential: k = 2000, p = 1e+07, rate = 0.001, task_id = 339474
30	POLICY ID = 0, task_id = 354932
31	distribution = boundedexponential: k = 2990, p = 1e+07, rate = 0.1, task_id = 358868
32	POLICY ID = 0, task_id = 374935
33	distribution = boundedexponential: k = 2900, p = 1e+07, rate = 0.01, task_id = 378923
34	POLICY ID = 0, task_id = 394939
35	distribution = boundedweibull: alpha = 1.5, p = 1e+07, beta = 3323.2, task_id = 400301
36	POLICY ID = 0, task_id = 414942
37	distribution = boundedexponential: k = 2990, p = 1e+07, rate = 0.1, task_id = 421038
38	POLICY ID = 0, task_id = 434947
39	distribution = boundedpareto: k = 28.5212, p = 1e+07, alpha = 0.7, task_id = 445606
40	CHANGE POLICY: POLICY ID = 1 CUT OFFS: 4156.32, 1e+07, task_id = 454947

Table 6.6: ADAPT-POLICY: Run 2, distribution change rate: very high

1	distribution = boundedpareto: k = 28.5212, p = 1e+07, alpha = 0.7
2	POLICY ID = 1 CUT OFFS: 84526.6, 1e+07, task_id = 70004
3	distribution = boundedexponential: k = 2000, p = 1e+07, rate = 0.001
4	distribution = boundedweibull: alpha = 0.5, p = 1e+07, beta = 1500
5	CHANGE POLICY: POLICY ID = 0, task_id = 90058
6	POLICY ID = 1 CUT OFFS: 6970.88, 1e+07, task_id = 110066
7	distribution = boundedweibull: alpha = 0.5, p = 1e+07, beta = 1500
8	POLICY ID = 1 CUT OFFS: 6504.32, 1e+07, task_id = 130090
9	distribution = boundedpareto: k = 28.5212, p = 1e+07, alpha = 0.7
10	POLICY ID = 1 CUT OFFS: 30125.5, 1e+07, task_id = 150091
11	distribution = boundedweibull: alpha = 2.5, p = 1e+07, beta = 3381
12	POLICY ID = 1 CUT OFFS: 17604, 1e+07, task_id = 170108
13	distribution = boundedpareto: k = 878.08, p = 1e+07, alpha = 1.4
14	POLICY ID = 1 CUT OFFS: 4168.56, 1e+07, task_id = 190116
15	distribution = boundedexponential: k = 2990, p = 1e+07, rate = 0.1
16	POLICY ID = 1 CUT OFFS: 3444.75, 1e+07, task_id = 210220
17	distribution = boundedweibull: alpha = 2.5, p = 1e+07, beta = 3381
18	CHANGE POLICY: POLICY ID = 0, task_id = 230225
19	distribution = boundedexponential: k = 2000, p = 1e+07, rate = 0.001
20	POLICY ID = 0, task_id = 250227
21	distribution = boundedexponential: k = 2990, p = 1e+07, rate = 0.1
22	POLICY ID = 0, task_id = 270228
23	distribution = boundedweibull: alpha = 2.5, p = 1e+07, beta = 3381
24	POLICY ID = 0, task_id = 290229
25	distribution = boundedpareto: k = 28.5212, p = 1e+07, alpha = 0.7
26	CHANGE POLICY: POLICY ID = 1 CUT OFFS: 34866.2, 1e+07, task_id = 310621
27	distribution = boundedpareto: k = 1571.53, p = 1e+07, alpha = 2.1
28	POLICY ID = 1 CUT OFFS: 18033.2, 1e+07, task_id = 330633
29	distribution = boundedexponential: k = 2000, p = 1e+07, rate = 0.001
30	CHANGE POLICY: POLICY ID = 0, task_id = 350756
31	distribution = boundedweibull: alpha = 2.5, p = 1e+07, beta = 3381
32	POLICY ID = 0, task_id = 370762
33	distribution = boundedpareto: k = 28.5212, p = 1e+07, alpha = 0.7
34	CHANGE POLICY: POLICY ID = 1 CUT OFFS: 40807.7, 1e+07, task_id = 390797
35	distribution = boundedweibull: alpha = 2.5, p = 1e+07, beta = 3381
36	POLICY ID = 1 CUT OFFS: 9710.09, 1e+07, task_id = 410937
37	distribution = boundedexponential: k = 2900, p = 1e+07, rate = 0.01
38	CHANGE POLICY: POLICY ID = 0, task_id = 431128
39	distribution = boundedpareto: k = 28.5212, p = 1e+07, alpha = 0.7
40	CHANGE POLICY: POLICY ID = 1 CUT OFFS: 3812.46, 1e+07, task_id = 451133

Table 6.7: Expected waiting for policies

Distribution change rate	Run no	Best policy
moderate	run 1	Random
moderate	run 2	ADAPT-POLICY
high	run 1	ADAPT-POLICY
high	run 2	ADAPT-POLICY
very high	run 1	ADAPT-POLICY
very high	run 2	STATIC-TAGS-LOW-VAR

Chapter 7

Discussion

This thesis looked at four important research questions related to scheduling and task assignment in server farms. These questions were formulated taking into consideration the major limitations associated with existing scheduling and task assignment policies. The solutions were proposed under two realistic workload scenarios, namely, the heavy-tailed service time distributions and arbitrary service time distributions. The key contributions of this thesis are summarised below.

- How can we efficiently schedule tasks in a time sharing server under heavy-tailed service time distributions?

  This research question deals with the way tasks are assigned in a time sharing server under heavy-tailed service time distributions. Our main focus was on multi-level time sharing policy (MLTP). The use of MLTP can result significant performance improvements over other policies, if the service time distribution of tasks possesses the property of decreasing failure rate, a key property of modern heavy-tailed traffic.

  Existing work on MLTP has been carried out under very unrealistic conditions such as the infinitely small quanta, infinite number of levels and exponential service time distributions. In contrast, we investigated the performance of MLTP under realistic conditions, i.e. under finite number of levels, when the quanta are not infinitely small, where the service time distributions are heavy-tailed. This model is more consistent with those implemented on real computer systems. The key findings are as follows.

  – We showed that MLTP with optimal quanta (MLTP-O) significantly outperforms both MLTP with equal quanta (MLTP-E) and FCFS, especially when the system load and the variability of service times are high. For example, under high system loads and high

task size variabilities, 2-MLTP-O (i.e. MLTP-O with two queues) outperforms FCFS and 2-MLTP-E (i.e. MLTP-E with two queues) by factors of 3 and 2 respectively. Under high system loads and moderate task size variabilities, 2-MLTP-O outperforms FCFS and 2-MLTP-E by factors of 2 and 1.5 respectively.

- We showed that as the number of levels increases the performance (i.e. the expected waiting time and expected slowdown) of both MLTP-O and MLTP-E increases. Moreover, the rate at which the performance increases depends on the variability of service times and the system load.
- We investigated the behaviour of quanta for 2-MLTP-O. We showed that the (optimal) set of quanta is unique for most of the scenarios. Moreover, for some system loads and task size variabilities, there is another set of quanta that will result in near optimal performance. When the expected waiting time is used (as the performance metric), we noted that there is a sudden drop in optimal Quantum 1 that occurs between the system loads of 0.5 and 0.7.
- We investigated the degradation in the expected waiting time for 2-MLTP-O, when quanta/cut-offs are computed to optimise the expected slowdown and vice versa. We showed that under high system loads and high task size variabilities (i.e. low α values), the degradation in the expected waiting time is very high when the quanta for 2-MLTP-O are computed to optimise the expected slowdown. For example, under very high system loads and very high task size variabilities, the degradation in the expected waiting time is equal to 250%. On the other hand, the degradation in the expected slowdown lies in the range 10%- 60% for all system loads and task size variabilities considered. In general, using relatively small values for Quantum 1 improves both the expected slowdown and expected waiting time. However, the use of very small values for Quantum 1 to optimise the expected slowdown can result in the expected waiting time to deteriorate significantly (250%).
- We investigated the performance of N-MLTP-E under a large number of queues using the expected waiting time and expected slowdown. We showed that the relationship between the performance and the number of levels has a power law relationship, and the coefficients of the power curve are functions of both the variability of tasks and the system load.
- We compared the performance of N-MLTP-E with the performance of FB and showed that under highly variable traffic conditions, N-MLTP-E requires a large number of queues

if it is to achieve the same performance levels as FB.

- How can we efficiently assign tasks in time sharing server farms under heavy-tailed workload conditions?

  This research question aimed at addressing a core issue associated with existing task assignment policies, which is their inability to efficiently assign tasks in time sharing server farms under heavy-tailed service time distributions.

  We extended the work proposed for the first research question, and proposed three task assignment policies dedicated to time sharing server farms. They are called: MLMS, MLMS-M and MLMS-PM. MLMS is suitable for time sharing server farms that do not have task migration facilities. On the other hand, MLMS-PM and MLMS-M are designed for time sharing server farms that support preemptive migration and non-preemptive migration respectively. MLMS improves the performance by giving preferential treatment to small tasks and by reducing the task size variability in queues within hosts. MLMS-M and MLMS-PM improve the performance by giving preferential treatment to small tasks and by reducing the task size variability in queues both within hosts (locally) and at host level (globally). For each task assignment policy, we provided an analytical performance model. The key finding related to these policies are summarised below.

  - MLMS: MLMS outperforms TAGS under certain workload scenarios, while TAGS outperforms MLMS under certain other scenarios. Whether or not MLMS outperforms TAGS depends on a number of factors, which include the number of levels, number of hosts, task size variability (i.e. α) and the system load. We noted that MLMS generally performs better than TAGS under high system loads in specific ranges of α, where α represents the task size variability. On the other hand, under low and moderate system loads, TAGS outperforms MLMS, particularly if the task size variability is high.

    The number of hosts does not affect the performance of MLMS under a given task size variability, system load and number of levels. This is because an increase/decrease in the number of hosts does not affect the service time distribution of tasks seen by hosts or the average arrival rates of tasks into hosts.

    MLMS does not support task migration and as such MLMS does not kill tasks, nor does it restart them from scratch. This means that it does not generate excess load on the system and therefore, it scales well. However, the main issue with MLMS is that it only performs

well under specific workload scenarios, particularly under low system loads and low task size variabilities.

– MLMS-M: MLMS-M outperforms TAGS for almost all the cases considered. The highest improvement in the performance is seen under high system loads and very high task size variabilities (i.e. low α values). For example, in a 2 Host system, when both the system load and the task size variability are high, MLMS-M with two levels outperforms TAGS by a factor of 2.7. Under the same conditions, MLMS-M with five levels outperforms TAGS by a factor 6.75.

  Under a fixed system load, the expected waiting time for MLMS-M does not continuously improve with the decreasing task size variability. This is particularly the case under moderate and high system loads when the number of levels is relatively high. As the task size variability decreases, the expected waiting time decreases up to a minimum value and then it starts to increase. This is because there is an increase in the excess load with the decreasing task size variability, due to restarting tasks from scratch. This results in the expected waiting time to degrade.

  Under a fixed system load, the expected waiting time for MLMS-M improves with the number of hosts for certain task size variabilities. For example, under moderate system loads and moderate task size variabilities, MLMS-M with three levels performs 1.5 times better in a 3 Host system compared to that of a 2 Host system. However, if the task size variability is less than a certain value, an increase in the number of hosts does not result in an improvement in the expected waiting time. Although a 3 Host system offers higher reduction in the variance of task sizes in queues, under low task size variabilities, a three hosts MLMS-M system generates significantly higher excess load compared to that of a 2 Host MLMS-M system. As a result, the performance of MLMS-M degrades in 3 Host system if the task size variability is low.

– MLMS-PM: MLMS-PM address the main issue associated with MLMS-M, i.e. poor performance under high α values and high system loads. MLMS-PM is based on pre-emptive task migration (work-conserving migration) and as such it does not restart tasks from scratch.

  The performance analysis of MLMS-PM shows that it outperforms TAGS-PM under all the scenarios considered. The highest improvement in the performance is obtained under high system loads and high task size variabilities. For example, in a 2 Host system, when both the system load and task size variability are high, MLMS-PM with five levels

outperforms TAGS-PM by a factor of 4.

We note that MLMS-PM outperforms MLMS under all the system loads considered. MLMS does not outperform MLMS-PM even if the number of levels of MLMS is very high. This means that two level (global and local) variance reduction model and preemptive task migration can result in significant performance improvements.

- How can we efficiently assign tasks in multiple batch server farms under heavy-tailed workload conditions?

  Our aim was to look into one of the key issues in existing task assignment policies, i.e. their inability to efficiently assign tasks in multiple server farm environments.

  First we proposed MTTMEL policy for a stand-alone batch server farm. This policy addresses the major issues in existing traditional task assignment policies (e.g. poor performance under high task size variabilities) as well as those of TAGS (e.g. poor performance under low and moderate task size variabilities, poor performance under high system loads and poor performance in large-sized server farms). MTTMEL is based on a flexible multi-tier host architecture that reduces the variance of tasks in hosts queues (in each tier). In addition, it speeds up the flow of small tasks by processing them in a relatively large number of hosts. These features allow MTTMEL to perform well under a range of heavy-tailed workload conditions. The number of tiers and number of hosts for MTTMEL are computed to optimise the performance under a given scenario. Second we extended MTTMEL and proposed MCTPM for assigning tasks in multiple server farms. MCTPM is based on the same multi-tier host architecture. MCTPM has the ability to control the traffic flow into server farms via a global dispatching device so as to optimise the performance. In addition, MCTPM supports preemptive task migration between servers in the same farm as well as between servers in different farms. This ensures that MCTPM can resume the execution of a task that was previously suspended at a different host.

  We provided an analytical performance models for MTTMEL and MCTPM taking into account the cost of migration and then carried out an extensive analytical and experimental performance analysis of policies under a wide range of workload conditions. The key finding relating to MTTMEL and MCTPM are as follows.

  - MTTMEL: This policy outperforms Random under a wide range of task size variabilities. The most significant improvements in the performance is seen when α lies in the range 0.7-1.5. For example, under a system load of 0.5, when α equals 1.1, MTTMEL outperforms Random by a factor of 23. In addition, MTTMEL outperforms TAGS under a

range of α values, particularly moderate and high α values. For example, under a system load of 0.7, when α equals 1.3, MTTMEL outperforms TAGS by a factor of 2.6. We note that MTTMEL operates in steady state throughout the entire range of α values considered. TAGS, on the other hand, fails to operate in steady state under a range of α values. The reason is that TAGS generates significantly large amounts of excess load compared to that of MTTMEL. For example, the ratio between TAGS excess and MTTMEL excess is equal to 4.6 under high α values and moderate system loads.

– MCTPM: We showed that MCTPM outperforms existing policies under a wide range of workload conditions. The experimental and analytical performance analysis of MCTPM in small-sized server farms demonstrates that this policy significantly outperforms MC-Random, MC-TAGSPM and MC-MTTPM. For example, under high system loads, high task size variabilities and low migration cost, MCTPM outperforms MC-Random by a factor of 190. Under moderate system loads, low task size variabilities and high migration cost, MCTPM outperforms MC-TAGSPM by a factor of 5. In small-sized server farms, MCTPM does not perform well if the system load and migration cost are high, and the variability of task sizes is low. In large-sized server farms, this specific case can be avoided by changing the number of tiers and the number of hosts allocated to tiers.

  MC-TAGSPM cannot operate in steady state (due to high excess load) under a range of conditions (e.g. high system loads and high task size variabilities). MCTPM does not suffer from this problem because it can control the fraction of tasks assigned to server farms, number of hosts and number of tiers in such a way that the system does not become unstable.

  We noted that the proportional migration cost has a significant impact on the performance if the task size variability is very high. However, under low and moderate task size variabilities, proportional migration cost does not impact the performance at large. The fixed migration cost has minimal effects on the performance under almost all the scenarios considered.

  We investigated the performance of MCTPM in large-sized server farms. We noticed that when there are a large number of hosts in the system, there is more than one possible host architecture for MCTPM. The architecture that results in the least expected waiting time depends on the system load and the task size variability.

- How can we efficiently assign tasks in server farms when the service time distribution of tasks is not known *a priori*?

The previous three research questions were related to performance optimisation under heavy-tailed workloads, while this research question investigated a way of designing adaptive task assignment policies that make no assumptions regarding the underlying service time distribution of tasks. We proposed an adaptive task assignment policy called, ADAPT-POLICY, which is based on a set of static-based task assignment policies. ADAPT-POLICY adaptively changes the task assignment policy to suit the most recent traffic conditions.

ADAPT-POLICY consists of three main stages, namely, the on-line data collection, on-line density estimation and on-line selection of task assignment policies. This policy aims at defining a set of static-based task assignment policies for a server farm and then utilising the task assignment policy with the least expected waiting time for assigning the next batch of tasks. The task assignment policy with the least expected waiting time is determined using the service time distribution of tasks as well as the various properties of the service time distribution that are estimated on-line using the service times of tasks collected over a period of time. Since ADAPT-POLICY adaptively changes the task assignment policy based on the most recent traffic conditions, it performs well under a range of evolving (non-stationary) traffic conditions. We developed a simulation model for ADAPT-POLICY using C++ based OMNET++ network simulator and investigated the performance of ADAPT-POLICY under a range of scenarios.

The key finding relating to ADAPT-POLICY are as follows.

- We showed that transformation-based non-parametric kernel density estimators can be used to estimate PDF and CDF of different types of distributions (with different properties) by applying these estimators to estimate PDF and CDF of Bounded Pareto, Bounded exponential and Bounded Weibull distributions. When estimating CDF and PDF, we noted that bandwidth has a significant effect on the quality of the fit and the accuracy of results. The bandwidth (i.e. h_n) for the density estimator is computed using the formula $h_n = n^{-r}$, where n denotes the sample size and $0.2 < r < 1$. Most of the r values, produced a good overall fit, in particular when estimating the cumulative distribution function. However, the r value that results in the most accurate fit varies from distribution to distribution and varies for different parameters for the same distribution. As such, ADAPT-POLICY estimates the service time distribution multiple times using different r values and then computes the expected waiting time for each policy (in the policy list) for each r. For each task assignment policy (in the policy list), the final expected waiting time is computed by taking the average of these individual expected waiting times.
- We investigated the moving average of waiting time for ADAPT-POLICY under three

different distribution change rates, namely, the moderate, high and very high. We noted that ADAPT-POLICY outperforms other task assignment policies (i.e. ADAPT-TAGS, Random and STATIC-TAGS) under most of the scenarios considered, because unlike other policies, ADAPT-POLICY can adaptively change the task assignment policy online based on the most recent traffic conditions. However, in a few scenarios ADAPT-POLICY does not perform so well because there is a delay in identifying the rapid changes in the incoming service time distribution. As a result, a non-optimal task assignment policy is used for assigning tasks for a short period of time, which results in a rapid increase in the moving average of waiting time for a short period of time. In addition, we compared the overall expected waiting time for ADAPT-POLICY with four other policies for six different cases. Out of these six cases that we considered, ADAPT-POLICY outperforms other policies in four cases.

7.1 Future work

The performance analysis provided in MLTP in Chapter 3 assumes that the context switch overhead of tasks can be equated to zero. In the cases where there is a significant context switch overhead, the context switch overhead should be taken into account when computing the optimal quanta for MLTP. Future work could consider incorporating the context switch overhead into the model and then investigating the effect of context switch overhead on the performance of MLTP.

MLMS-PM model proposed in Chapter 4 assumes that tasks possess minimal state information and therefore, migration cost is negligible. It is possible for certain types of tasks to incur significant amounts of migration cost on the system. In this case, these costs will have be to incorporated into the performance model. This could be considered in future work. However, it is anticipated that inclusion of these costs would result in very complex analytical models for time sharing systems.

MCTPM proposed in Chapter 5 supports preemptive migration of tasks between server farms. When a task is migrated from a particular host in one server farm to another host in a different server farm, it is possible that there is some delay between the time at which the task arrives at the destination host and the time at which the task departs the source host. MCTPM assumes that this delay is at its minimal level and hence can be equated to zero. In order for this condition to be true, it may be necessary for the set of server farms to be located in close proximity to each other. In the case where server farms are located far apart, there are significant delays and additional delay parameters will have to be included into the performance equations to cater for such delays. This we did not consider in MCTPM. Future work could look into incorporating the delay components into

the analytical model and then investigating the effect of these delays on the performance.

In Chapter 6, we noted that ADAPT-POLICY could result in promising results under a range of workload scenarios even under rapidly changing service time distributions. The future work could consider the following two enhancements to improve the computational time of ADAPT-POLICY.

1. Distribution comparison test

   The current version of ADAPT-POLICY does not perform any tests to check if there is a significant difference between the previous service time distribution and the current service time distribution prior to computing the optimal performance for policies. The distribution comparison test compares the current estimated distribution with the previously estimated distribution and if the two distributions are not significantly different then the same policy will be used to assign the next batch of tasks. To compare two distributions, methods such as Kolmogorov-Smirnov [Mood et al., 1974] test can be used.

2. Learning algorithm

   When ADAPT-POLICY operates in exhaustive mode, it estimates the service time distribution of tasks and its distributional properties for each new batch of tasks, which completes their processing. The purpose of the learning algorithm is to learn how fast the service time distribution of tasks changes (i.e. distribution change rate) over time. Once this rate is computed then ADAPT-POLICY can reduce the frequency of the density estimation. For example, if the rate at which distribution changes is low, then there is no need to perform density estimation and optimisation exhaustively rather it can be done in a periodic manner (for example every 10 hours, $1,000,000$ tasks, etc.). If there are no changes in the service time distribution then there is no need to perform on-line density estimation at all. In this case one static-task assignment policy can be used to assign tasks with the correct scheduling parameters. If the system appears to be performing poorly then the density estimation procedure can be reinitiated.

Bibliography

S. Aalto. M/G/1/MLPS compared with M/G/1/PS within service time distribution class IMRL. *Mathematical Methods of Operational Research*, 64(2):309–325, October 2006.

S. Aalto, U. Ayesta, and E. Nyberg-Oksanen. Two-level processor-sharing scheduling disciplines: mean delay analysis. *In Proceedings of the Joint International Conference on Measurement and Modelling of Computer Systems, New York, USA*, 32(1):97–105, June 2004.

S. Aalto, U. Ayesta, and E. Nyberg-Oksanen. M/G/1/MLPS compared to M/G/1/PS. *Operations Research Letters*, 33(5):519–524, September 2005.

S. Aalto, U. Ayesta, S. Borst, V. Misra, and R. Núñez-Queija. Beyond processor sharing. *ACM SIGMETRICS Performance Evaluation Review*, 34(4), March 2007.

C. Amza, A. Cecchet, S. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic content benchmarks. *In Proceedings of the Fifth IEEE Workshop Workload Characterization*, November 2002.

M. Arlitt and T. Jin. A workload characterization study of the 1998 world cup website. *IEEE Network*, 14(3):30–37, May/June 2000.

M. Arlitt and C. L. Williamson. Web server workload characterization, the search for invariants. pages 126–138, May 1996.

M. F. Arlitt and C. L. Williamson. Internet Web servers: workload characterization and performance implications. *IEEE/ACM Transactions on Networking*, 5:631–645, October 1997.

K. Avrachenkov, U. Ayesta, P. Brown, and E. Nyberg. Differentiation between short and long tcp flows: predictability of the response time. *In IEEE Infocom 2004 Hong Kong*, pages 762–773, May 2004.

J. Balasubramanian, D. C. Schmidt, L. Dowdy, and O. Othman. Evaluating the performance of middleware load balancing strategies. In *EDOC '04: Proceedings of the IEEE International Conference on Enterprise Distributed Object Computing*, pages 135–146, Washington, DC, USA, 2004. IEEE Computer Society.

P. Barford and M. E. Crovella. Generating representative Web workloads for network and server performance evaluation. pages 151–160, June 1998.

P. Barford, A. Bestavros, A. Bradley, and M. E. Crovella. Changes in Web client access patterns: Characteristics and caching implications. *World Wide Web, special issue on characterization and performance evaluation*, 2:15–28, 1999.

D. Bertsimas and G. Mourtzinou. Transient laws of non-stationary queueing systems and their applications. *Queueing Systems, Theory and Applications*, 25:115–155, January 1997.

S. C. Borst, M. Mandjes, and M. J. G. Van Uitert. Generalised processor sharing queues with light-tailed and heavy-tailed input. *IEEE/ACM Transactions on Networking*, 11:821–834, June 2003.

J. Broberg, Z. Tari, and P. Zeephongsekul. Task assignment based on prioritising traffic flows. *Conference on Principles of Distributed Systems*, pages 15–17, December 2004.

J. Broberg, Z. Tari, and P. Zeephongsekul. Task assignment with work-conserving migration. *Parallel Computing*, 32(11-12):808–830, 2006.

R. Cáceres, P. B. Danzig, S. Jamin, and D. J. Mitzel. Characteristics of wide-area TCP/IP conversations. SIGCOMM '91, pages 101–112, New York, NY, USA, 1991. ACM.

J. Cao, W. S. Cleveland, and D. X. Sun. On the nonstationarity of Internet traffic. *ACM Performance Evaluation Review*, 29(1):102–112, June 2001.

O. Cappe, E. Moulines, J.-c. Pesquet, A. P. Petropulu, and X. Yang. Long-range dependence and heavy-tail modeling for teletraffic data. *IEEE Signal Processing Magazine*, 19(3):14–27, May 2002.

V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu. The state of the art in locally distributed Web-server systems. *ACM Computing Surveys*, 34(2):263–311, 2002.

N. Chen and S. Jordan. Throughput in processor-sharing queues. *IEEE Transactions on Automatic Control*, 52(2):299–305, 2007.

H. Cheng, N. Xia, and Y.-q. Fang. SVM classification of heavy-tailed Internet traffic. *Huadong Ligong Daxue Xuebao/Journal of East China University of Science and Technology*, 36(6):807–811, 2010.

E. Chlebus and R. Ohri. Estimating parameters of the Pareto distribution by means of Zipf's law: application to Internet research. In *Global Telecommunications Conference*, volume 2, pages 5–28, Nov.-2 Dec. 2005.

G. Ciardo, A. Riska, and E. Smirni. EQUILOAD: a load balancing policy for clustered Web servers. *Performance Evaluation*, 46(2-3):101–124, 2001.

E. G. Coffman and L. Kleinrock. Feedback queueing models for time-shared systems. *Journal of the ACM*, 15(4):549–576, October 1968.

M. E. Crovella and A. Bestavros. Self-similarity in World Wide Web traffic: evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5:835–846, December 1997.

M. E. Crovella, M. Harchol-Balter, and C. Murta. Task assignment in a distributed system: Improving performance by unbalancing load. pages 268–269, June 1998a.

M. E. Crovella, M. S. Taqqu, and A. Bestavros. *Heavy-tailed probability distributions in the World Wide Web*, pages 3–25. Birkhauser Boston Inc., Cambridge, MA, USA, 1998b.

E. de Souza e Silvia and M. Gerla. Queueing network models for load balancing in distributed systems. *Journal of Parallel and Distributed Computing*, 12(1):24–38, 1991.

S. Dhakal, M. Hayat, J. Pezoa, C. Yang, and D. Bader. Dynamic load balancing in distributed systems in the presence of delays: A regeneration-theory approach. *IEEE Transactions on Parallel and Distributed Systems*, 18(4):485–497, April 2007.

A. Di Stefano, L. Lo Bello, and E. Tramontana. Factors affecting the design of load balancing algorithms in distributed systems. *Journal of Systems and Software*, 48(2):105–117, 1999.

M. Dobber, R. van der Mei, and G. Koole. Dynamic load balancing and job replication in a global-scale grid environment: A comparison. *IEEE Transactions on Parallel and Distributed Systems*, 20(2):207–218, February 2009.

A. B. Downey. Lognormal and pareto distributions in the Internet. *Computer Communications*, 28: 790–801, 2005.

A. B. Downey. Evidence for long-tailed distributions in the Internet. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, IMW '01, pages 229–241, New York, NY, USA, 2001. ACM.

J. F. Epperson. *An Introduction to Numerical Methods and Analysis*. John Wiley and Sons, 2001.

D. G. Feitelson and M. A. Jette. Improved utilization and responsiveness with gang scheduling. In *IPPS '97: Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 238–261, London, UK, 1997. Springer-Verlag.

D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *IPPS '97: Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 1–34, London, UK, 1997. Springer-Verlag.

H. Feng and V. Misra. Mixed scheduling disciplines for network flows. *ACM Performance Evaluation Review*, 31(2):36–39, 2003.

B. Fu and Z. Tari. A dynamic load distribution strategy for systems under high task variation and heavy traffic. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 1031–1037, New York, NY, USA, 2003. ACM.

M. Gray. *Growth and Usage of the Web and the Internet*, 2009. http://www.mit.edu/~mkgray/net/.

D. Gross and C. M. Harris. *Fundamentals of Queueing Theory*. Wiley Series in Probability and Statistics, 1998.

M. M. Group. *World Internet Users and Population Stats*, 2009. http://www.internetworldstats.com/stats.htm.

M. Harchol-Balter. Task assignment with unknown duration. *In Proceedings of the Twentieth International Conference on Distributed Computing Systems*, pages 214–224, 2000.

M. Harchol-Balter. Task assignment with unknown duration. *Journal of the ACM*, 49(2):260–288, 2002.

M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15:253–285, August 1997.

M. Harchol-Balter, M. Crovella, and C. D. Murta. On choosing a task assignment policy for a distributed server system. *Journal of Parallel and Distributed Computing*, 59(2):204–228, 1999.

M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-based scheduling to improve Web performance. *ACM Transactions on Computer Systems*, 21(2):207–233, 2003.

M. Ibrahim and L. Xinda. Performance of dynamic load balancing algorithm on cluster of workstations and pcs. In *Proceedings of the Fifth International Conference on Algorithms and Architectures for Parallel Processing*, pages 44–47, 2002.

E. Isogai. The convergence rate of fixed width sequential confidence intervals for a probability density function. *Sequential Analysis*, 6(1):55–69, 1987.

Q. j. Lin, D. Chen, and Y. c. Liu. Non-stationary and small-time scaling behavior of Internet traffic. In *Proceedings of International Conference on Communications, Circuits and Systems*, volume 3, pages 1717–1721, june 2006.

M. Jayasinghe, Z. Tari, and P. Zeephongsekul. The impact of quanta on the performance of multi-level time sharing policy under heavy-tailed workloads. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, ACSC '09, pages 95–104. Australian Computer Society, Inc., 2009a.

M. Jayasinghe, Z. Tari, P. Zeephongsekul, and J. Broberg. On the performance of multi-level time sharing policy under heavy-tailed workloads. In *IEEE Symposium on Computers and Communications*, pages 956–962, July 2009b.

M. Jayasinghe, Z. Tari, and P. Zeephongsekul. Multi-level multi-server task assignment with work-conserving migration. In *Ninth IEEE Symposium on Network Computing and Applications*, pages 178–181, 2010a.

M. Jayasinghe, Z. Tari, and P. Zeephongsekul. A scalable multi-tier task assignment policy with minimum excess load. In *IEEE Symposium on Computers and Communications*, pages 913–918, 2010b.

M. Jayasinghe, Z. Tari, and P. Zeephongsekul. Performance analysis of multi-level time sharing task assignment policies on cluster-based systems. In *IEEE International Conference on Cluster Computing*, pages 265–274, 2010c.

M. Jayasinghe, Z. Tari, P. Zeephongsekul, and A. Y. Zomaya. Task assignment in multiple server farms using preemptive migration and flow control. *Journal of Parallel and Distributed Computing*, 71:1608–1621, December 2011.

M. C. Jones, J. S. Marron, and S. J. Sheather. A brief survey of bandwidth selection for density estimation. *Journal of the American Statistical Association*, 91(433):401–407, 1996.

T. Karagiannis, M. Molle, and M. Faloutsos. Long-range dependence: Ten years of Internet traffic modeling. *IEEE Internet Computing*, 8(5):57–64, 2004.

D. G. Kendall. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain. *The Annals of Mathematical Statistics*, 24(3):338–354, 1953.

J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948, nov/dec 1995.

L. Kleinrock. *Queuing Systems Volume I*. John Wiley and Sons, 1975.

L. Kleinrock. *Queueing Systems, Volume II: Computer Applications*. John Wiley and Sons, New York, 1976.

L. Kleinrock and R. R. Muntz. Processor-sharing queueing models of mixed scheduling disciplines for time-shared systems. *ACM*, 19(3):464–482, July 1972.

B. Krogfoss, G. Hanson, and R. Vale. Impact of consumer traffic growth on mobile and fixed networks: Business model and network quality impact. *Bell Labs Technical Journal*, 16(1):105–120, 2011.

W. Leland and T. J. Ott. Load-balancing heuristics and process behavior. *Performance Evaluation Review*, 14:54–69, May 1986.

R.-c. Liu. Analysis of the effects of system parameters on load balancing. *Information Sciences Informatics and Computer Science: An International Journal*, 81(1-2):37–54, 1994.

P. Loiseau, P.-b. Primet, and P. Goncalves. A long-range dependent model for network traffic with flow-scale correlations. *Stochastic Models*, 27(2):333–361, 2011.

S. Luo and G. A. Marin. Simulating application level self-similar network traffic using hybrid heavy-tailed distributions. In *ACM-SE 43: Proceedings of the Forty-third annual Southeast regional conference*, pages 54–58, New York, NY, USA, 2005. ACM.

K. U. Markovich, N.M. Analyzing measurements from data with underlying dependences and heavy-tailed distributions. pages 425–436, 2011.

A. Menon. *Frontiers of Evolutionary Computation*, volume 11. Springer, 2004.

D. S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, 2000.

M. Mitzenmacher. A brief history of generative models for power law and lognormal distributions. *Internet Mathematics*, 1:226–251, 2004.

A. M. Mood, F. A. Graybill, and D. C. Boes. *Introduction to the Theory of Statistics*. McGraw-Hill, Singapore, third edition, 1974.

M. Nuyens and A. Wierman. The foreground-background queue: A survey. *Performance Evaluation*, 65(3-4):286–307, 2008.

A. M. Odlyzko. Internet traffic growth: Sources and implications. *Optical Transmission Systems and Equipment for WDM Networking II*, pages 1–15, 2003.

E. W. Parsons and K. C. Sevcik. Implementing multiprocessor scheduling disciplines. In *IPPS '97: Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 166–192, London, UK, 1997. Springer-Verlag.

J. Philip and A. Rasch. Queueing theory study of round-robin scheduling of time-shared computer systems. *Journal of the ACM*, 17(1):131–145, January 1970.

R. Poli, J. Kennedy, and T. Blackwell. Particle swarm optimization. *Swarm Intelligence*, 1:33–57, 2007.

K. Psounis, P. Molinero-Fernández, B. Prabhakar, and F. Papadopoulos. Systems with multiple servers under heavy-tailed workloads. *Performance Evaluation*, 62(1-4):456–474, 2005.

I. Rai, G. Urvoy-Keller, and E. Biersack. Analysis of LAS scheduling for job size distributions with high variance. *In: Proceedings of ACM Sigmetrics, San Diego, USA*, pages 218–228, 2003.

I. Rai, E. Biersack, and G. Urvoy-Keller. LAS scheduling approach to avoid bandwidth hogging in heterogeneous TCP networks. pages 179–190, 2004a.

I. Rai, G. Urvoy-Keller, M. Vernon, and E. Biersack. Performance analysis of LAS-based scheduling disciplines in a packet switched network. *In ACM SIGMETRICS/PERFORMANCE 2004, New York*, pages 106–117, 2004b.

I. Rai, E. Biersack, and G. Urvoy-Keller. Size-based scheduling to improve the performance of short TCP flows. *IEEE Network Magazine*, 19(1):12–17, January 2005.

R. Righter and J. G. ShanthiKumar. On external service disciplines in single-stage queueing systems. *Journal of Applied Probability*, 27:409–416, 1990.

A. Riska, W. Sun, E. Smirni, and G. Ciardo. ADAPTLOAD: effective balancing in clustered Web servers under transient load conditions. pages 104 – 111, 2002.

M. Sakata, S. Noguchi, and J. Oizumi. An analysis of the M/G/1 queue under round-robin scheduling. *Operations Research*, 19(2):371–385, March-April 1971.

L. E. Schrage. The queue M/G/1 with feedback to lower priority queues. *Management Science*, 13 (7):466–474, 1967.

W. Shu and J. Wang. Min-min chromosome genetic algorithm for load balancing in grid computing. *International Journal of Distributed Sensor Networks*, 5(1):62–63, 2009.

A. Silberschatz, P. B. Galvin, and G. Gagne. *Applied Operating Systems Concepts*. John Wiley and Sons, Addison-Wesley Publishing Company, 1998.

B. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman and Hall, 1986.

J. S. Simonoff. *Smoothing Methods in Statistics*. Springer Series in Statistics, 1996.

M. Srinivas and L. Patnaik. Genetic algorithms: a survey. *Computer*, 27(6):17–26, June 1994.

Z. Tari, J. Broberg, A. Y. Zomaya, and R. Baldoni. A least flow-time first load sharing approach for distributed server farm. *Journal of Parallel and Distributed Computing*, 65:832–842, 2005.

B. Theophilus, A. Aditya, and M. Dave. Network traffic characteristics of data centers in the wild. In *Internet Measurement Conference 2010*, Melbourne, Australia, 2010.

V. Ungureanu, B. Melamed, and M. Katehakis. Effective load balancing for cluster-based servers employing job preemption. *Performance Evaluation*, 65(8):606–622, 2008.

A. Varg. The OMNeT++ discrete event simulation system. *In Proceedings of the European Simulation Multiconference*, June 2001.

M. Wand and M. Jones. *Kernel Smoothing*. Chapman and Hall, 1995.

R. R. Weber. On the optimal assignment of customers to parallel servers. *Journal of Applied Probability*, 15(2):406–413, 1978.

A. Wierman, N. Bansal, and M. Harchol-Balter. A note on comparing response times in the M/GI/1/FB and M/GI/1/PS queues. *Operations Research Letters*, 32(1):73–76, 2004.

C. Williamson. Internet traffic measurement. *IEEE Internet Computing*, 5:70–74, 2001.

W. Willinger, M. S. Taqqu, R. Sherman, and D. V. Wilson. Self-similarity through high-variability: statistical analysis of Ethernet LAN traffic at the source level. *In Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 25: 100–113, October 1995.

W. Willinger, M. S. Taqqu, R. Sherman, and D. V. Wilson. Self-similarity through high-variability: statistical analysis of Ethernet LAN traffic at the source level. *IEEE/ACM Transactions on Networking*, 5(1):71–86, 1997.

W. Winston. Optimality of the shortest line discipline. *Journal of Applied Probability*, 14(1):181–189, 1977.

Wolfram Research. Mathematica version 5.0. 2003.

S. F. Yashkov. Processor-sharing queues: some progress in analysis. *Queueing Systems, Theory and Applications*, 2(1):1–17, 1987.

Q. Zhang and W. Sun. Workload-aware load balancing for clustered Web servers. *IEEE Transactions on Parallel and Distributed Systems*, 16(3):219–233, 2005.

Q. Zhang, A. Riska, E. Riedel, and E. Smirni. Bottlenecks and their performance implications in e-commerce systems. *In Proceedings of the Ninth International Workshop Web Caching and Content Distribution*, pages 273–282, October 2004.

Z. Zhang and W. Fan. Web server load balancing: A queueing analysis. *European Journal of Operational Research*, 186(2):681–693, 2008.

Z.-L. Zhang, V. Ribeiro, S. Moon, and C. Diot. Small-time scaling behaviors of Internet backbone traffic: an empirical study. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 3, pages 1826–1836, 2003.

S. Zikos and H. D. Karatza. The impact of service demand variability on resource allocation strategies in a grid system. *ACM Transactions on Modeling and Computer Simulation*, 20:1–29, November 2010.