DYNAMIC MEMORY SCHEDULING TO ENHANCE PROCESSING PERFORMANCE

(UYGULAMA PERFORMANSINI ARTIRMAK İÇİN DİNAMİK HAFIZA PLANLAMASI)

by

Mutlu ERCAN, B.S.

Thesis

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

in the

INSTITUTE OF SCIENCE AND ENGINEERING

of

GALATASARAY UNIVERSITY

September 2009

DYNAMIC MEMORY SCHEDULING TO ENHANCE PROCESSING PERFORMANCE

(UYGULAMA PERFORMANSINI ARTIRMAK İÇİN DİNAMİK HAFIZA PLANLAMASI)

by

Mutlu ERCAN, B.S.

Thesis

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Date Submission : September 12, 2009

Date of Defense Examination : October 12, 2009

Supervisor : Asst. Prof. Dr. Tankut ACARMAN

Committee Members : Asst. Prof. Dr. Murat AKIN

Assoc. Prof. Dr. Esra ALBAYRAK

Acknowledgements

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. I want to thank the Computer Engineering Department at Institute of Science and Engineering in Galatasaray University for giving me permission to commence this thesis in the first instance, to do the necessary research work and to use departmental data.

I am deeply indebted to my supervisor Asst. Prof. Dr. Tankut ACARMAN from Galatasaray University whose help, stimulating suggestions and encouragement helped me in all the time of research for and writing of this thesis.

My colleagues from the Department of Technology Management at AvivaSA Emeklilik ve Hayat supported me in my work. I want to thank them for all their help, support, interest and valuable hints. Especially I am obliged to Altan EVNİ and Özge KUTAL for their great help in difficult times.

Especially, I would like to give my special thanks to my wife Nazlı whose patient love enabled me to complete this work.

Mutlu ERCAN

September 2009

Table of Contents

List of Symbols

t_r : Running time cost of a computational job.
t_c : Time cost of getting output of a computational from the memory
m : Total memory usage to keep output of the jobs
m_T : Total available memory
λ : Jobs arrival rate
α : Proportion of the memory usage
t_{CPU} : Time cost of a computational job
N : Number of jobs in a window
Δt_{CPU}^{k} : Total time cost of computational jobs in a window
$\Delta t_{I/O}^{k}$: Total time cost of I/O jobs in a window
a_{kj} : Existence of a computational job in a window
b_{ki} : Existence of an I/O job in a window
T_{Total} : Total time of the processing jobs in a window
e_k : Error value
v_d : Desired value
v_k : Intermediate variable
g_k^l : Achieved gain
CPU : Central Processing Unit
ILC : Iterative Learning Control
μ : Service rate
λ : Arrival rate
p^{sc} : Selective criteria
τ : Window size
C : Capacity of the memory
Ψ_i : Individual gain
Ω : The cumulative number of the individual distinct jobs

List of Figures

List of Tables

Abstract

In line with the increasing information processing requests of users and systems, the systems with much more powerful processing capabilities are needed. Despite the increase in hardware's processing and transmission busses' speed, performance enhancement of the overall computing system may require simultaneous resource allocation depending on the computing task. To enhance the overall processing performance, research on software control and dynamic resource allocation of the computing and storage units have been underway by almost all the leading computer and computing nodes' manufacturers. In this study, computing system performance enhancement by dynamic memory scheduling has been developed.

A fairly special computing process has been elaborated, the requested computing jobs are always created by choosing the inputs among a finite set inquiring the risk of the insurance policy of the candidate. In some special computing areas, like insurance risk investigation, calculations of income and premium need heavy and repetitive actuarial calculations, that the computing systems performing these computing efforts, may enhance CPU's utilization by classifying on an "intelligent manner" the computing jobs and caching the repeating computing jobs created by the same input interval. In this study, caching the repetitive jobs' results and enhancing the CPU usage rate has been presented and an intelligent regulation scheme has been introduced to software control.

The jobs can be classified as two kinds; I/O jobs and computational jobs. External storage jobs are may be the examples of the I/O jobs. The computational jobs are the jobs, which always produce the same outputs versus the same input values. In this work, in the situation that the jobs are processed again; the advantages are examined to keep the outputs in the memory instead of reprocessing them.

Instead of a static structure, to keep the process in a dynamic way to provide more benefit from the memory has investigated, the Iterative Learning Control methodology has been implemented to improve the usage of the system resources. Moreover, theoretical results, which obtained in this work has observed on the actual practice. The data that has been obtained from an individual retirement company has been corrupted by a linear function.

Résumé

Aujourd'hui, en conformité avec les besoins croissants des utilisateurs et des systèmes, nous avons besoin des systèmes qui sont plus puissants. En dépit de l'augmentation de la vitesse de matériel, là peut surgir quelques problèmes de performance dans les systèmes s'ils ne sont pas appui par le logiciel. Afin d'empêcher ces problèmes, les travails sur la couche application a fait. Dans cette étude, nous explorons la manière d'augmenter les performances système par l'établissement du programme dynamique de mémoire.

Dans le cas nous recherchons les systèmes ; nous notons que quelques travaux ont retraité encore. Nous pouvons classifier les travaux de système comme travaux d'entrée-sortie et travaux informatiques. Les travaux de stockage externe sont peuvent être les exemples des travaux d'entrée-sortie. En outre, les travaux informatiques sont les travaux, qui produisent toujours les mêmes sorties par les mêmes valeurs d'entrée. Dans ce travail, dans la situation cette les travaux traitent encore ; nous avons examiné les avantages de maintenir les sorties dans la mémoire au lieu de les retraiter.

Au lieu d'une structure statique, pour maintenir le processus dans une manière dynamique de fournir plus d'indemnité de la mémoire a étudié. D'ailleurs, les résultats théoriques, qui ont obtenu en ce travail a observé sur la pratique réelle.

Özet

Günümüzde artan kullanıcı ve sistem ihtiyaçları doğrultusunda daha güçlü sistemlere ihtiyaç duyulmaktadır. Artan donanım hızlarına rağmen yazılım tarafından desteklenmeyen sistemlerde performans sorunları ortaya çıkabilmektedir. Uygulama katmamnında bu amaç doğrultusunda bazı çalışmalar sürekli olarak yapılmaktadır. Bu çalışmada, dinamik hafıza planlaması ile sistem performansını arttırmanın yollarını araştırıldı.

Sigorta sektörü gibi bazı sektörlerde prim ve kazanç hesaplamaları için yoğun aktueryal hesaplamalar yapılmaktadır. Bu hesaplamaları yapan programlarda da, her ne kadar farklı girdi değerleri ile işlemler yapılsa da, sistemin alt katmanlarında bazı işlerin sürekli olarak tekrarlandığı görülebilmektedir. Her işlem esnasında bütün hesaplamaların baştan yapılmasındansa, çok tekrar gören işlerin sonuçlarını sistemin performansını düşürmeden hafızada tutmanın faydaları araştırılmıştır.

Sistem işlerini de I/O işi ve hesap işi olarak sınıflandırabiliriz. I/O işlerine dış depolama aygıtları üzerinde yapılan işleri örnek olarak verebiliriz. Hesap işleri ise aynı girdi değerleri ile daima aynı çıktıları üreten işlerdir. Çalışmamızda hesap işlerinin tekrarlandıkları durumda tekrar tekrar işlemek yerine çıktı değerlerini bellekte tutmanın avantajları incelenmiştir.

Bellekte tutma işlemi için durağan bir yapı yerine dinamik olarak bellekten daha fazla faydalanmanın yolları araştırılmıştır. Elde edilen teorik sonuçların gerçek uygulamalar üzerindeki etkisi gözlemlenmiştir.

Öncelikle sanal veri setleri üzerinde algoritma denendi. 10 tane iş, normal dağılım ile rastgele olarak 10000 iş oluşturacak şekilde dizildi. Herbir iş için rastgele çıktı

büyüklüğü ve servis hızı seçildi. Farklı büyüklükteki hafıza durumları için bu işler tekrarlandı. İş sayısı 100'e kadar onar onar arttırılarak sonuçlar gözlemlendi. Elde edilen kazançlar, öğrenme algoritması uygulanmadığı takdirde elde edilebilecek kazançlar ile karşılaştırıldı.

Daha sonra bu sonuçlar ışığında bir bireysel emeklilik şirketinin, hayat sigortası ürünleri için prim hesaplayan, bireysel emeklilik ürünleri için de birikim projeksiyonu yapan "Zaman Makinası" adındaki, Java altyapısı ile yazılmış uygulamasının çalışma istatistiklerinden oluşan gerçek data üzerinde çalıştırılarak, öğrenme algoritması uygulanmadığı takdirde oluşan sonuç ile sınızsız hafızaya sahip olunması durumunda elde edilebilecek sonuçlar ile karşılaştırıldı. Bireysel emeklilik şirketinden alınan data, doğrusal bir fonksiyon vasıtasıyla bozulmuştur.

Bu çalışmada temel olarak çoğu zaman boşta kalan hafıza tekrarlayan işler için, çıktı değerlerini tutmak üzere kullanılması incelenmiştir. Dataların hafızada tutulması işlemi tasarım modeli olarak kullanılmakta olan bir yapıdır. Bu modelin işler için de kullanılabilirliği araştırılmıştır. İş tekrarlı yapılarda, uygun iş seçimi için eklenen öğrenme algoritması ile birlikte sonuçları gözlemlenmiştir.

1. Introduction

Nowadays, by the increase of the users and the demands of the business, more powerful systems are required. Despite of the development speed of hardware systems, it has to be assisted by software systems, because incomplete performance work at software layer can cause slowness even crashes. In the application layer, some improvements are developed to enhance computing performance. In this sense, efficient usage of system resources is a vital part towards process' development.

One of the main resources of the system is CPU and another one is memory. Using them efficiently makes the system works faster. The system can need them all sometimes, but usually some of the system resources can be inactive. Mostly, the system cannot use all bytes of the memory. Because of that, we studied to find how the usage of the memory could be augmented without decreasing the performance of the system.

To achieve this goal, we tried to improve the performance of the Time-Machine application of an individual retirement and life insurance enterprise which is developed on Java platform. Java class methods are used as jobs and the heap space is tried to be used efficiently. The application calculates the premium and the retention depends on the customer's age, sex and assurance. First of all, the application calculates a multiplicity depends on the age and sex of the customer, and then the multiplicity is multiplied by the premium or the assurance. Java methods are recalled with same parameters during the day. The idle memory is tried to be allocate for the recalling methods.

2. Literature Review

In order to improve system performance, many studies has been developed on different subjects including scheduling strategies, architectures and device improvements. The materials which compose devices have been enhanced to run faster. Nowadays faster, multi-cored, multi-CPU systems are being produced, faster bus' are being used to connect their processors with larger and faster memories, faster and larger external storage devices has been developed.

Many static and dynamic scheduling algorithms have been developed for real time system' operations. Static scheduling algorithms have low costs, but incapable to adapt scheduling rules to enhance efficiency of the limited resources. Dynamic scheduling algorithms provide more adaptive solutions to the systems to respond the changes in the environment systems [1]. The dynamic scheduling algorithms mainly based on "selection criteria driven" approach such as Earliest Deadline First, Highest Value First, Highest Value Density First.

The Earliest Deadline First Algorithm's selective criterion is based on the deadlines of task's current requests. A task will be assigned the highest priority if the deadline of its current request is the nearest, and will be assigned the lowest priority if the deadline of its current request is the furthest [2]. In The Highest Values First Algorithms, each process has its own function of time which defines the value to the system of that process. The completion of a process has a value to the system which varies with time [3]. The Highest Value Density First Algorithm analyzes the dynamic value density of a task during its runtime at the first time. Both the value and time attributes of a task are considered when assigning its priority [4].

On the other hand, static scheduling algorithms have been improved by adding them dynamic approach. Improved round-robin algorithm with two processors based on separating the processors by process type, one is dedicated for CPU-intensive process, and the other one is for I/O dedicated process [5].

Data access operations occupy much in system resources. One of the design patterns in software development architecture to improve system performance is caching. The applications avoid from repeated data reads by caching. The cache strategies depend on requirements of the applications. There are several cache patterns as below [6];

- Cache Accessor - Decouples caching logic from the data model and data access details.
- Demand Cache - Populates a cache lazily as applications request data. A demand cache is useful for data that is read frequently but unpredictably.
- Primed Cache - Explicitly primes a cache with a predicted set of data. A primed cache is useful for data that is read frequently and predictably.
- Cache Search Sequence - Inserts shortcut entries into a cache to optimize the number of operations that future searches require.
- Cache Collector - Purges entries whose presence in the cache no longer provides any performance benefit.
- Cache Replicator - Replicates operations across multiple caches.
- Cache Statistics - Record and publish cache and pool statistics using a consistent structure for uniform presentation.

Caching is one of the common performance improvement patterns which are applied in many layers of the systems. Figure 2.1 shows some caching implementations [7].

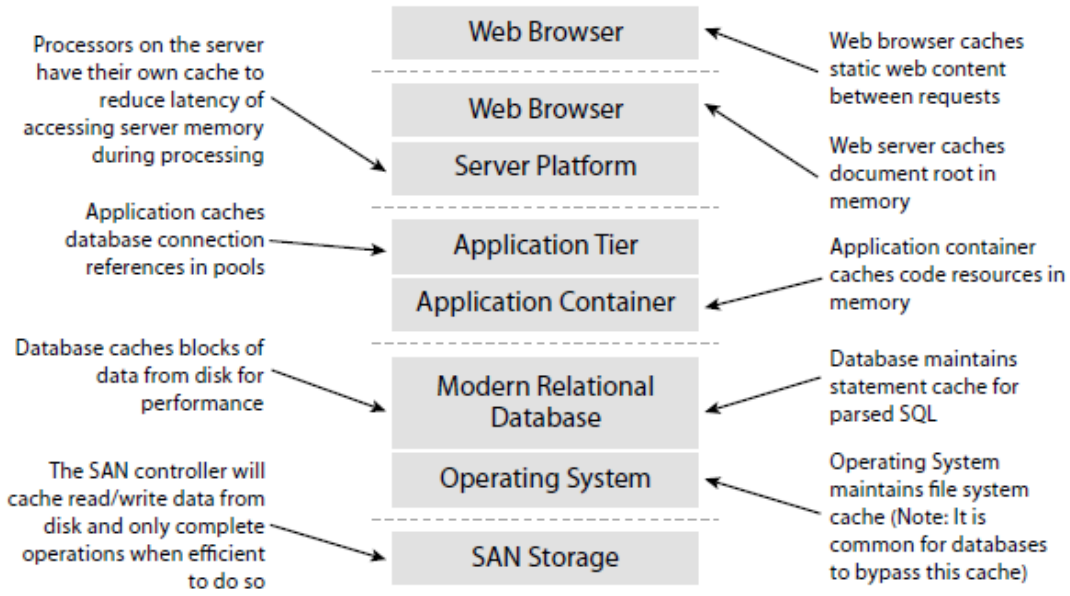


Figure 2.1 Caching exist in many system implementations [7].

Iterative Learning Control is a methodology which can be applied to repetitive systems tries to improve system performance based on the knowledge of the previous experiences. System learns from previous repetitions to improve next repetitions to minimize the tracking error [8]. Since Iterative Learning Control methodology (ILC) was submitted in 1984, the concept has been used in many areas. ILC has been applied in two dimensional systems with splitting the system into separated one dimensional system [9]. In factories, industrial manipulators performs same cycles repetitively, by using ILC some improvements in energy, time minimization has been gained [10]. ILC-based automatic train operation systems has been developed to converge the tracking error to zero which compose from energy savings, trip time, safety [11]. ILC has been used in antilock braking systems in electric and hybrid electric vehicles by using electric motor to improve the braking torque, where anti-lock brake systems learn the maximum tire-road friction, which may vary with the weather conditions such as rain, snow, dry etc, and maximize braking force by enforcing learning the braking actuators to search the maximum handling force, [12].

Since the submission of ILC, many studies have been performed on the subject. The convergence speed tried to be improved [13]. Studies on initial conditions have been performed in [14].

In this study, a memory scheduling methodology is explored. Idle memory tried to be used for caching repetitive jobs and ILC was tried to be applied to select proper job.

2.1. Motivation

In some areas, like insurance and retirement enterprises area, the calculations of retention and premium need heavy actuarial calculations. Like different types of mortality tables, disability, assurance, commission calculation tables, funds grow expectations and many other ones are used to perform these calculations. The tables are not changed frequently. The tables that are used for life insurance calculations, are updated when death proportions or born rates are changed (i.e. a disaster or an epidemic disease occurs). Individual retirement tables are changed when enterprise's strategy changes [15]. It can be assumed that for a life insurance policy with same input values always same premium is calculated.

Every day more than 10000 calculations are done. For an individual retirement product, a customer must be more than 18 years old and 56 years old customer can retire, and must be in the system more than 10 years. Most of customers are between 18 and 46. About 350 customers exist in each age. About 175 of them are male and 175 of them are female. For all types of calculations, for each customer a multiplicity calculating and for retirement products premium is multiplied by that multiplicity and for life insurance products assurance values are multiplied by that multiplicity. Many calculations are executed for each request coming from different individual customer. The Time Machine's customer information entry page is shown in Figure 2.2., an individual retirement products entry page can be seen in Figure 2.3 and a life insurance products entry page can be seen in Figure 2.4. The Time Machine is explained in section 4.1.

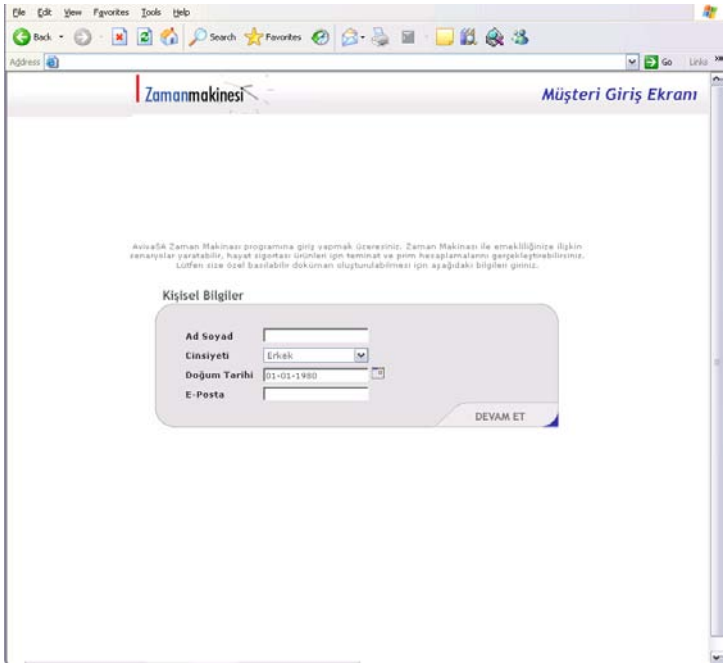


Figure 2.2 Time Machine customer information entry interface, [16]

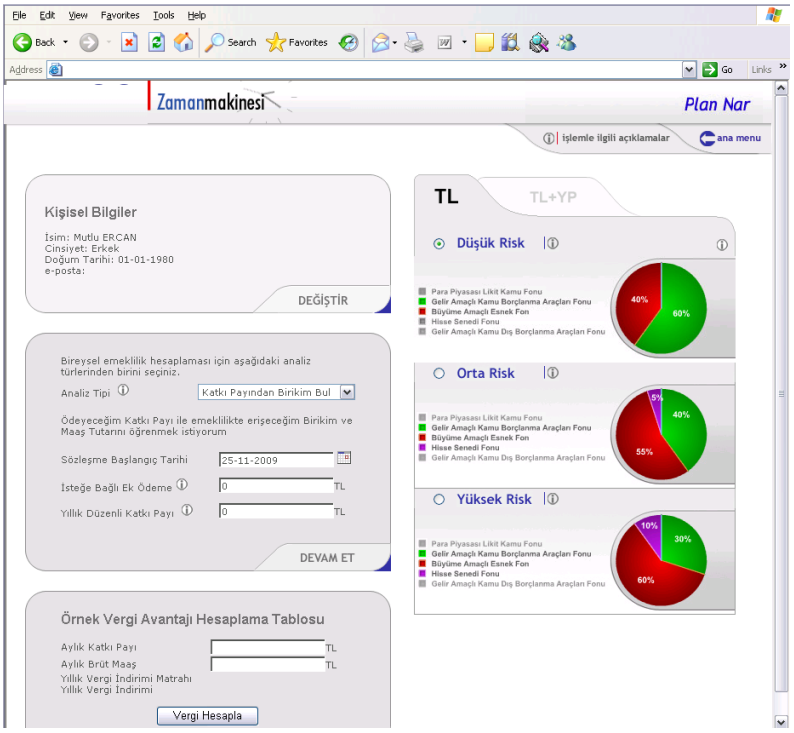


Figure 2.3 Time Machine an individual retirement product's property selection interfaces, [16]

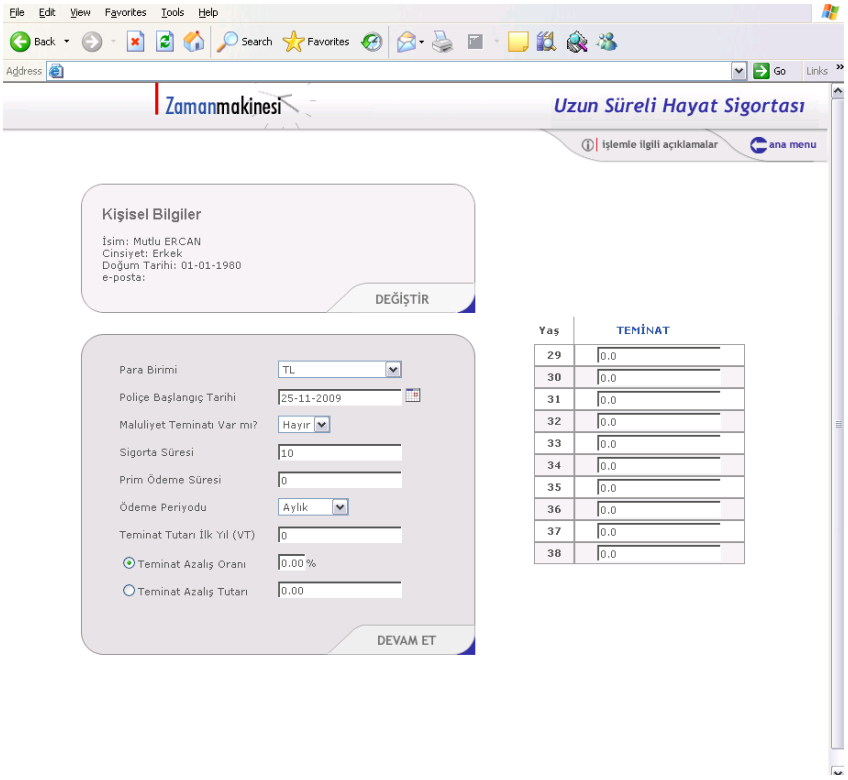


Figure 2.4 Time Machine a life insurance product's assurance property selection interfaces, [16]

The objective of the study is to improve system performance by establishing Dynamic Memory Allocation methodology. Our goal is to use the system memory in a maximum efficient way. To achieve this goal, the proposed resource allocation algorithm based on ILC to select the most profitable job has been developed which uses the system memory to preserve time.

One of the main parts of the system is the CPU and another one is the memory. All jobs operate in CPU by using the system memory. Software application developers evolve caching methods for frequently recalled jobs in order to ease work and shorten the cycle. In this way, frequently called jobs, which are specified by the architect, are kept

in the memory after their first call they have been used and which will be used from memory storage for their next calls. The success of this method depends on the experience of the developer. If he/she forces so many jobs kept within the memory, the memory demands for another job that may not be covered due to the system limits.

3. Computing Process Modeling

A computing system generally has six main parts. In some systems, some of the parts would be unnecessary and some of them would be more than one. Input values arrive from input devices by users or other systems. Depends on the CPUs availability the requests which are sent by input device can wait in the job queue. CPU performs the jobs in the job queue by using, if necessary, memory and external storage devices, and sends the result to the output device which can be a display device or another system or something else. The architecture of the system is shown in Figure 3.1.

The jobs come to the system randomly. Input devices indicate the devices that the parameters of the jobs are given. Output devices indicate the devices to which the output of the job will be sent. FIFO (First in first out) rule is applied to manage the job queue.

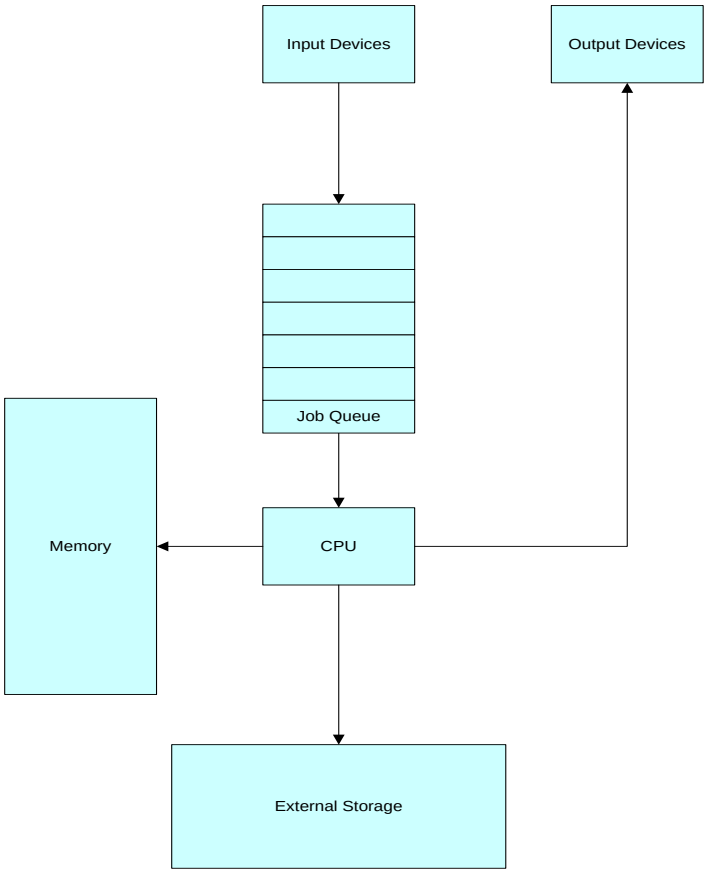


Figure 3.1 The architecture of the computing system

3.1. Memory

One of the parts of a system is the memory which is used to store data or programs and variables. In our scope, we try to use idle memory to keep the jobs' output values. Accessing to the memory for the CPU is fast enough. Because of that more output kept in the memory, provides the total time cost of the system in smaller degrees.

In our research, we obtained the average time of getting an output from memory is approximately 12*10e-7 ms. A priori, cost of a simple request from the memory is smaller than complex memory usages.

3.2. CPU (Central Processing Unit)

Another vital part is the CPU which can be called as core of the system because of the where all jobs should be processed. The CPU uses the memory to perform its jobs. Sometimes, for simple jobs CPU does not need to use the memory. This case does not cause a problem; it is detailed in Section 3.4. It can be approved that CPU is very busy and that is why we try to decrease its density.

3.3. Job

Jobs are whatever done by the CPU. But in this paper, jobs are defined as the expressions in the programming languages. 3+5 is a job. The jobs have the inputs and the outputs and these outputs can be used as an input of another job or can be assigned to a variable. The assignment operations cannot be defined as a job, x=8 is not a job.

There are two kinds of jobs; I/O jobs and computational jobs. The computational jobs are the jobs which are deterministic, give always same output with same input. “3+x” is a computational job, because when a value is set for “x” the output is independent from time. “3+4” equals always 7. The other jobs, like writing to a file system are I/O jobs.

3.4. The Process Modeling

Jobs that run in the systems can be categorized as I/O and the computational jobs. I/O jobs are performed by using system resources such as hard disk, ram. The computational jobs are the jobs that produce the same output with the same inputs. An I/O job such as data writing to a disk is not a deterministic job from this point of view. In some systems, the computational jobs are called frequently. If the frequencies of these jobs are so overly, the effect of caching them in the memory would achieve a better system performance than running them again.

We assume that the jobs which produce the same output by the same input are marked with same identity number. By the way, if a computational job, which's output kept in the memory, is recalled through, its output can be obtained from the memory.

The time cost of a job which runs in CPU can be called as t_r and the time cost of getting its output from memory is denoted by t_c. The computing process addressed by this study performs the process subject to the jobs with $t_r >> t_c$. It can be supposed if there are more jobs in the memory, there will be less time cost in the system. In this research, there is a massive difference between t_r and t_c. Getting from memory is a simple operation which is due t_c, but to run a complex job CPU uses the other devices such as memory and external devices several times.

Nowadays the CPUs are faster, it means the gain can be limited. But the gain can be improved by caching these jobs in groups.

If m describes the total memory footprints of the outputs of the jobs, and also m_T describes the total memory of the system.

$$\alpha = \frac{m}{m_T} \tag{3.1}$$

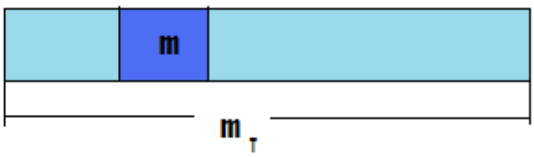


Figure 3.2 Memory Proportion

The equation (3.1) defines the proportion of the memory to cache the jobs and Figure 3.2 shows the memory proportion. If α is bigger for a homogeneous system, it means

more number of the jobs exist in the memory. And also the possibility of finding a job in the memory will be higher. But it may cause another problem; to run other jobs system may require the memory which is used to keep the outputs of the previous jobs. As a result, these jobs may run slower and the performance of the system may decrease. In other words, the jobs whose outputs exist in the memory are affected positively by some α. But for the jobs whose output does not exist in the memory, may run slower, because of the limited memory. Our goal is to develop an Iterative Learning Control System that achieves to equilibrate these two constraints and caching suitable jobs without increasing the overall service time.

An application that operates I/O and computational jobs randomly was prepared to observe the gain and this study enforces selection of operation points achieving higher gains. Figure 3.3 demonstrates time costs with respect to the classified job types. When most of the jobs are I/O jobs, there is a limited gain. But when the proportion of the computational jobs increase versus the overall jobs, the effect of keeping output of the jobs in the memory increases also. The left slope of the valley shows that effect. But when the system becomes unable to find enough memory to process the jobs, time costs of the jobs increases as shown in the right slope of the valley.

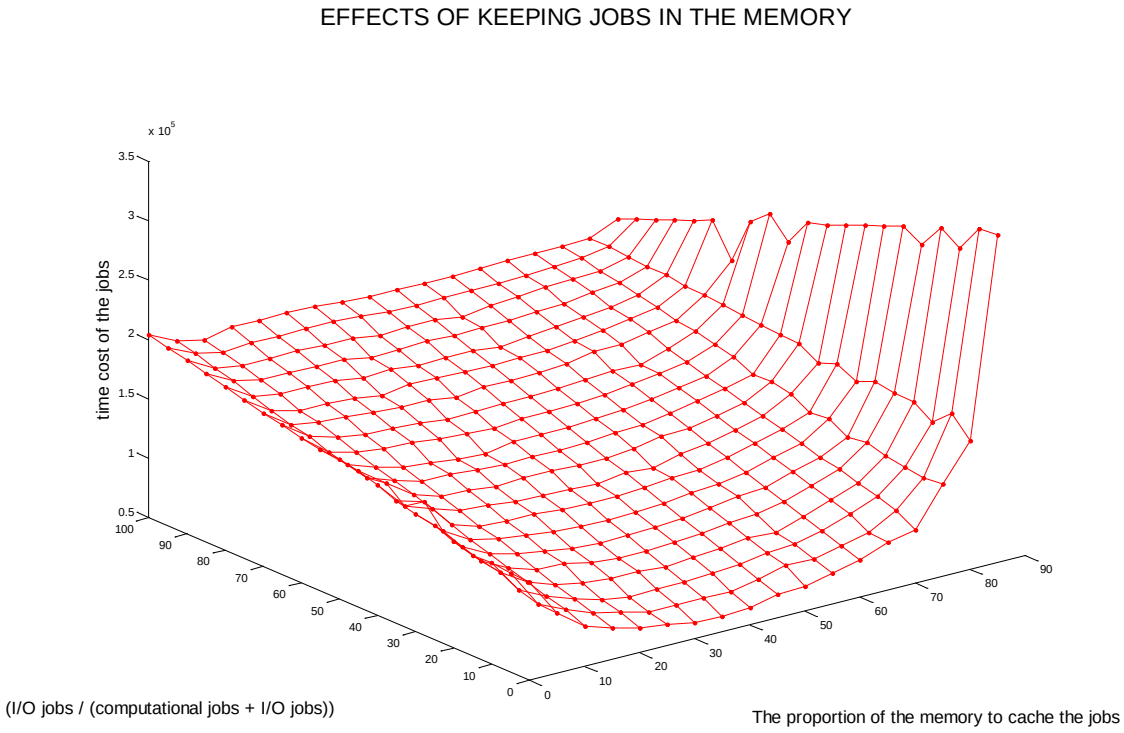


Figure 3.3 Effects of keeping the jobs in the memory

These jobs are homogenous in the sense of service rates and output size.

The time cost of a computational job can be describe as,

$$t_{CPU}(\alpha) = \begin{cases} t_c & \text{if the job exists in the memory} \\ t_r & \text{if the job is operated by the CPU} \end{cases} \quad (3.2)$$

Where sub indices c denotes cache and r denotes runtime. The jobs are observed by sliding window method. Initially, the last n jobs can be supposed as the current window size. After termination of a job, that job will enter to the window and the earliest job in the window will be out of the window. The total operating time of the k^{th} window is shown as;

$$\Delta t_{CPU}^{k} = \sum_{i=1}^{m} b_{ki} t_{CPU}^{ki}(\alpha) \qquad \forall a_{kj}, b_{ki} \in \{0,1\} \tag{3.3}$$

Where b_{ki} denotes the existence of i[th] job in the k[th] window. And the total time cost of the I/O jobs in the same window is;

$$\Delta t_{I/O}^{k} = \sum_{j=1}^{m} a_{kj} t_{I/O}^{kj}(\alpha) \qquad \forall a_{kj}, b_{ki} \in \{0,1\} \tag{3.4}$$

Where a_{kj} denotes the existence of the j[th] job in the k[th] window.

The jobs taken from I/O unit and to be computed existing in the window equals to;

$$T_k(\alpha) = \Delta t_{CPU}^{k} + \Delta t_{I/O}^{k} \tag{3.5}$$

In the equation (3.5), T_k defines the total time cost of the k[th] window, n defines the number of I/O jobs and m defines the number of computational jobs. $\Delta t_{I/O}^{k}$ denotes the time cost of the j.th I/O jobs that is being operated in the k[th] window.

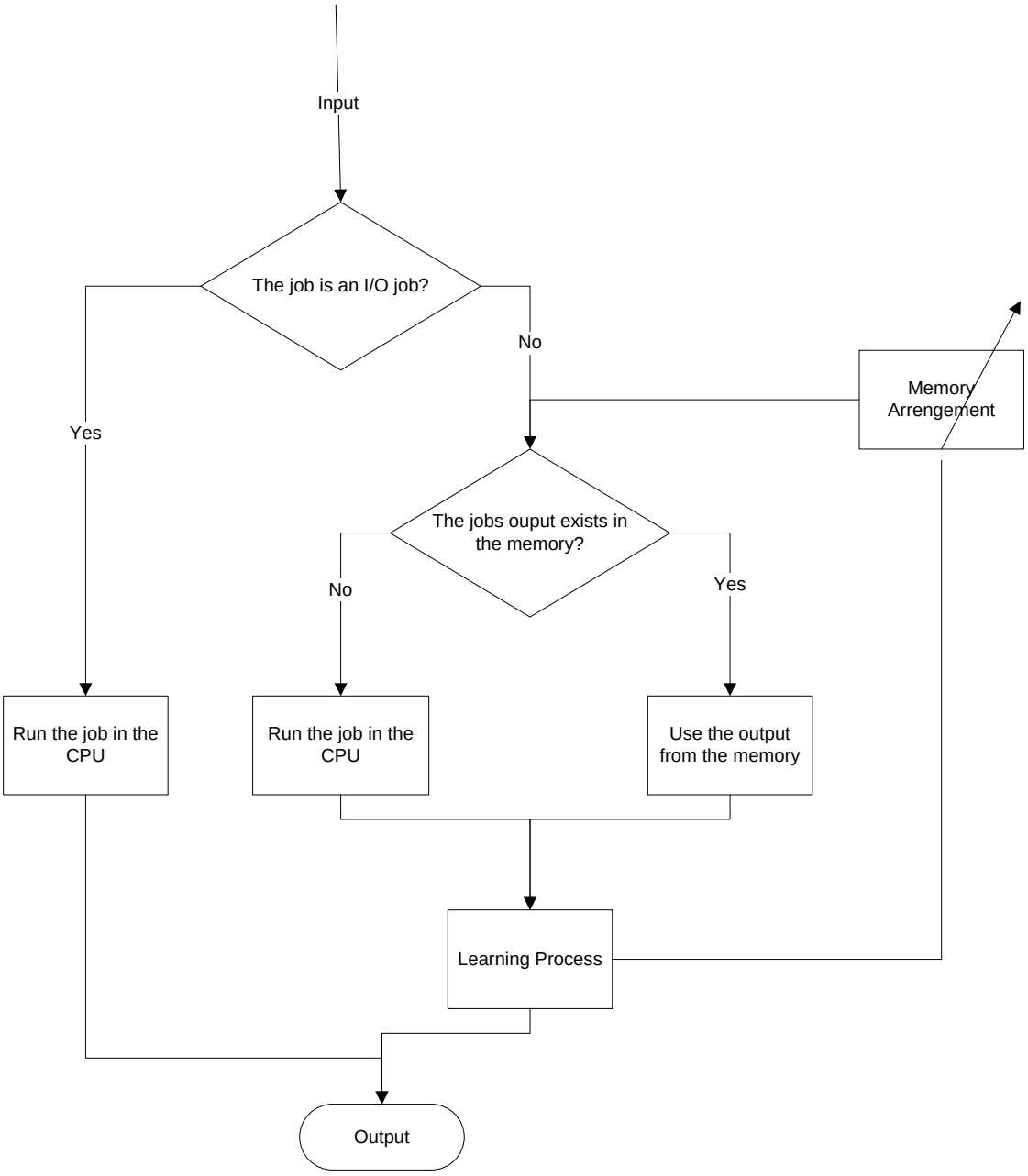


Figure 3.4 Flow diagram of the computing process

In the Figure 3.4, the fundamental process of the system is shown. The system decides to run a job by the type of the job and existence of the job in the memory. The interactions between CPU and Memory are shown in Figure 3.5.

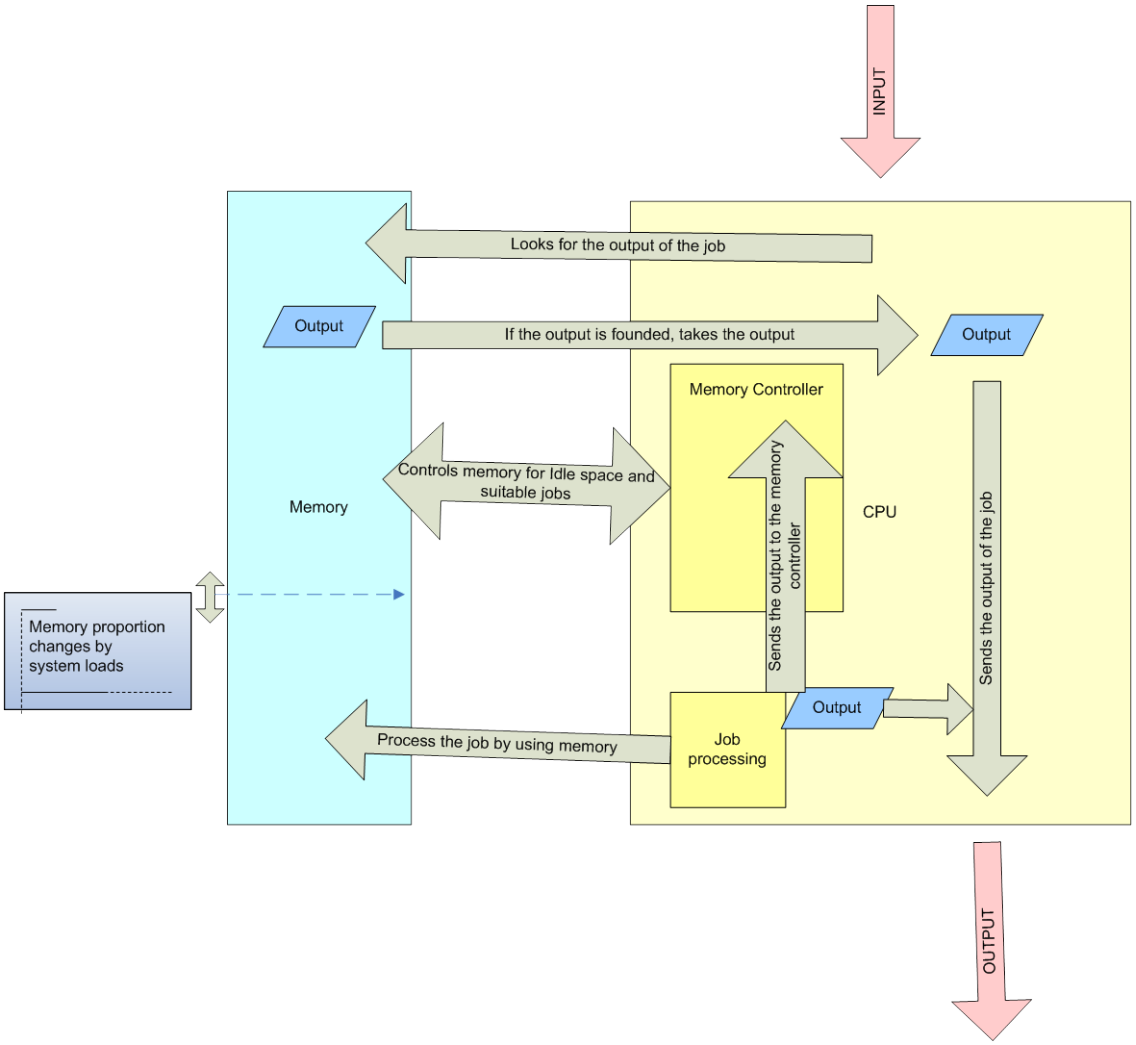


Figure 3.5 Interactions between CPU and memory

The $M|M|1$ queuing system consists of a single queuing station with a single server. Customers arrive according to a Poisson Process with rate λ, and the probability distribution of the service time is exponential with mean $\frac{1}{\mu}$ sec [17].

The individual jobs come to the computing system with Poisson Process with rate λ_i. Through the definition of Poisson Process [17], the jobs are merged into a single Poisson Process with rate equals the sum of the rates of its components. The system can be modeled as $M|M|1$ system. The jobs arrive by the Poisson Process, for all t, $\delta > 0$ [17];

$$P(A(t+\delta)-A(t)=n)=e^{-\lambda\delta}\frac{(\lambda\delta)^n}{n!} \quad n=1,2,... \tag{3.6}$$

There are two possible paths for the jobs. If the output of the job is in the memory, the job goes through the path that the memory exists on. Otherwise the job goes through the path that the CPU exists on to process the job. Probability of existence in the memory of the output of the job is p. The model of the process is as Figure 3.6.

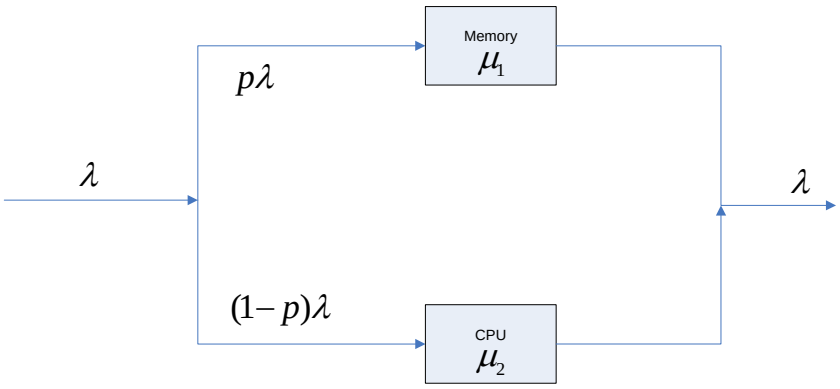


Figure 3.6 The model of the system

Where λ is the arrival rate of the jobs, μ_1 is the service rate of getting the output of the job from the memory and μ_2 is the service rate of processing the job by the CPU.

$$\mu_1 >> \mu_2 \tag{3.7}$$

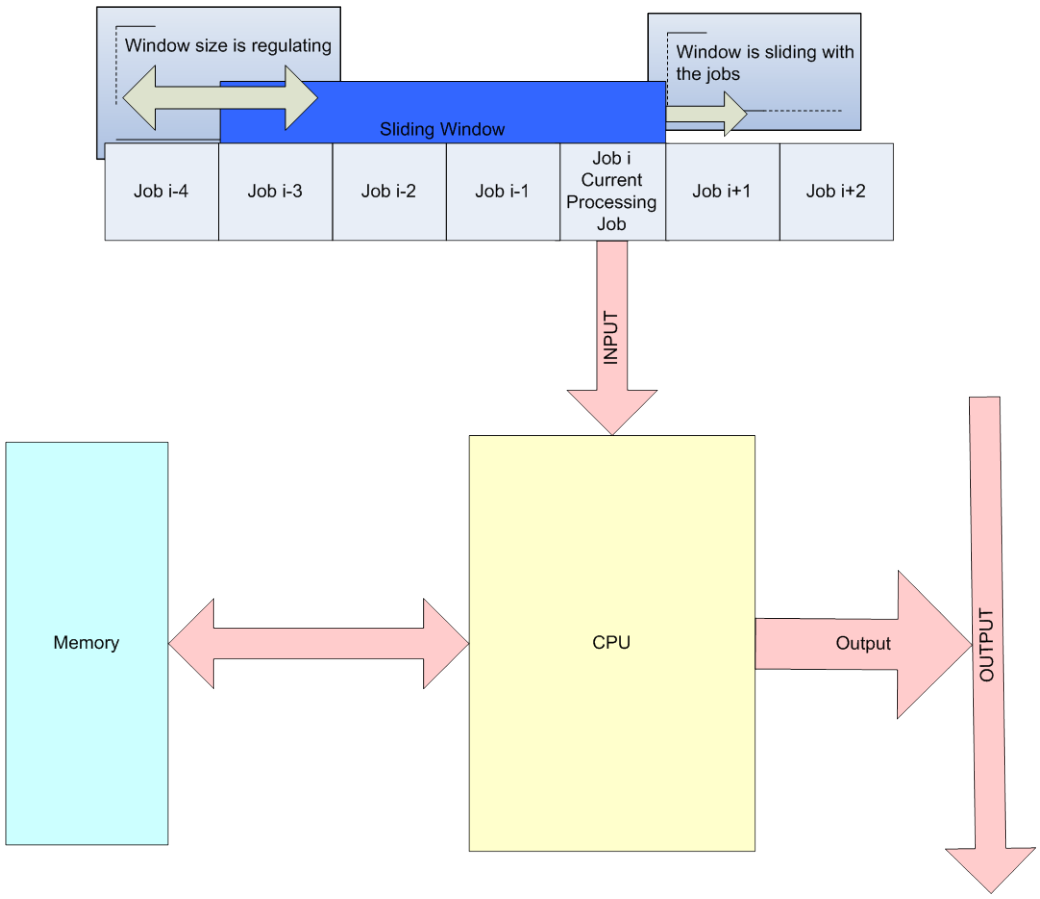


Figure 3.7 Sliding Window and window regulation process.

p^{sc}_i is the selection parameter to maintain the already computed result in the memory. The sliding window methodology can be used for forgetting the jobs having high selective criteria but which have not been recalled for a long time. The size of the sliding window is regulating to choose the most suitable jobs; the process is shown in Figure 3.7. The selective criteria are calculated for only the jobs that exist in the window. The jobs are sorted by their selective criteria multiplied par the number of the occurrences in the window, when there is enough memory the jobs put into the memory one by one. The selective criteria are as (3.8) and takes values between 0 and 1.

$$p^{sc}{}_i(\tau) = e^{-(\frac{a\frac{m_i}{C}}{\lambda_i \frac{\frac{1}{\mu_i}}{\sum_{i=1}^{\Omega}\frac{1}{\mu_i}}}\tau)} = e^{-(\frac{am_i\mu_i\Gamma\tau}{\lambda_i C})} \quad (3.8)$$

where $\Gamma = \sum_{i=1}^{\Omega}\frac{1}{\mu_i}$ and m_i is the size of the output of the i[th] job, μ_i is the service rate of the i[th] job by the CPU, Ω is the number of the jobs in the system, τ is the window dimension (number of jobs in the window), λ_i is the arrival rate of the i[th] job, C is the capacity of the memory and a is the scaling factor. We supposed that all of the variables of the jobs are determined *a priori*.

When m_i increases, $p^{sc}{}_i$ decreases. Because the job occupies more memory resource and it is better to chose with low m_i. When μ_i increases, $p^{sc}{}_i$ decreases. Because the time benefit decreases. When λ_i increases, $p^{sc}{}_i$ increases. It is favorable to choose the jobs with higher occurrence probability to avoid unnecessary computation efforts. When τ increases, $p^{sc}{}_i$ decreases. Because the numbers of the jobs increase, the importance of choosing the job decreases also.

There are two factors for a job to exist in the memory. One is existence in the window which is described by $\tilde{p}_i$, and the other one is selective criteria. The probability of existence in the memory (p_i) can be defined as (3.9).

$$p_i(\tau) = \tilde{p}_i(\tau)p^{sc}{}_i(\tau) \quad (3.9)$$

The probability of selecting a job from the system is number of occurrences of the job divided by the number of total jobs (3.10).

$$P(i) = \frac{\mathrm{N}(i)}{\sum_{j in System} \mathrm{N}(j)} \tag{3.10}$$

Where N(i) denotes number of occurrences of i^{th} job, and P(i) denotes the probability of selecting i^{th} job in the system. The probability of selecting another job is shown as (3.11).

$$\overset{*}{P}(i) = 1 - P(i) = 1 - \frac{\mathrm{N}(i)}{\sum_{j in System} \mathrm{N}(j)} = \frac{\mathrm{N}_{Total} - \mathrm{N}(i)}{\mathrm{N}_{Total}}$$

Where
$$\mathrm{N}_{Total} = \sum_{j in System} \mathrm{N}(j) \tag{3.11}$$

Where $\overset{*}{P}(i)$ is the probability of absence of job i in the sliding window, $P(i)$ is the probability of existence of job I, N is the total number of jobs. When we select another job, the probability of absence of the job from the two selections becomes as (3.12).

$$\overset{*}{P}_{12}(i) = \frac{\mathrm{N}_{Total} - \mathrm{N}(i)}{\mathrm{N}_{Total}} \frac{\mathrm{N}_{Total} - \mathrm{N}(i) - 1}{\mathrm{N}_{Total} - 1} \tag{3.12}$$

The probability of absence in a window for a job is as (3.13).

$$\overset{*}{P}_{\tau}(i) = \prod_{k=1}^{\tau} \frac{\mathrm{N}_{Total} - \mathrm{N}(i) - k + 1}{\mathrm{N}_{Total} - k + 1} \tag{3.13}$$

Where $\tilde{p}_i$ is the probability of existence of the i[th] job in the window as (3.14).

$$\tilde{p}_i(\tau) = 1 - \prod_{k=1}^{\tau} \frac{\mathrm{N}_{Total} - \mathrm{N}(i) - k + 1}{\mathrm{N}_{Total} - k + 1} \tag{3.14}$$

A(i) is the total number of i[th] job which can be described as the arrival rate of the proportion of i[th] job and the total arrival rate of the jobs multiplied with total number of jobs, described as (3.15).

$$\mathrm{N}(i) = \frac{\lambda_i}{\sum_{j=1}^{\Omega} \lambda_j} \mathrm{N} \tag{3.15}$$

The jobs for which CPU does not use the memory to process, the service rate tends to the infinite utilization rate, and the selective criteria would be neglected. These types of jobs would not be kept in the memory unless there is no other computational job which may be unrealistic scenario for real-time computing systems.

Gain of a job is defined as the time cost of CPU processing to achieve the output result. Time cost of getting from memory is ignored, because it is small and invariant (almost

same time cost value for all jobs). The relative gain is important for the system, the proportion of the absolute gain and the total of absolute gains of all jobs. The relative gain of the i[th] job in the window is denoted by Ψ_i, and given by,

$$\Psi_i = \frac{\frac{1}{\mu_i}}{\Gamma} = \frac{1}{\mu_i \Gamma} \qquad \text{Where} \qquad \Gamma = \sum_{i=1}^{\Omega} \frac{1}{\mu_i} \tag{3.16}$$

The expectation of a random variable X, is the mean of the distribution of X, denotes E(X) as given below:

$$E(X) = \sum_{all\ x} xP(X = x) \tag{3.17}$$

That is the average of all possible values of X, weighted by their probabilities [18]. For the total expected gain of a window, it can be obtained the possibilities of existence in the memory, weighted by the relative gains as (3.18).

$$E(\Psi) = \sum_{i\ in\ \tau} \Psi_i p_i(\tau) \tag{3.18}$$

From (3.7), (3.8), (3.13), (3.15), (3.17) we can obtain (3.19).

$$E(\Psi) = \sum_{i=1}^{\Omega} \left(\frac{1}{\mu_i \Gamma} \left(\left(1 - \prod_{k=1}^{\tau} \frac{N_{Total} - N(i) - k + 1}{N_{Total} - k + 1} \right) e^{-\left(\frac{am_i \mu_i \Gamma \tau}{\lambda_i C}\right)} \right) \right) \tag{3.19}$$

Ω stands for the cumulative number of the individual distinct jobs whose outputs are kept in the memory.

We suppose that the system has 3 jobs given by a, b and c which have the same output size, same service rate, and same arrival rate as given in Table 3.1 and variations of the expected values are given in Table 3.2. Total of the expected computing properties such as service rate, arrival rate and size may be maximized when the window size equals to 4, (see for instance Table 3.2 presenting the instantaneous dynamic resource allocation scheme given by (3.8) through (3.19).

Table 3.1 The characteristics of the jobs in the example

	a	b	c
μ_i (1/ms)	1	1	1
m_i (byte)	1	1	1
λ_i	1/3	1/3	1/3

Γ (ms)	3
Capacity(byte)	2

Table 3.2 Variations of the Expected Values

	Expected Values			
Window Size	a	b	c	Total
1	0,004240997	0,004240997	0,004240997	0,012722992
2	0,006588577	0,006588577	0,006588577	0,01976573
3	0,007682481	0,007682481	0,007682481	0,023047444
4	0,007969234	0,007969234	0,007969234	0,023907703
5	0,007757089	0,007757089	0,007757089	0,023271267
6	0,007255928	0,007255928	0,007255928	0,021767783
7	0,006606127	0,006606127	0,006606127	0,019818382
8	0,00589936	0,00589936	0,00589936	0,017698079
9	0,00519351	0,00519351	0,00519351	0,015580531
10	0,004523323	0,004523323	0,004523323	0,013569969
11	0,003907948	0,003907948	0,003907948	0,011723845
12	0,003356262	0,003356262	0,003356262	0,010068786
13	0,002870582	0,002870582	0,002870582	0,008611746
14	0,002449259	0,002449259	0,002449259	0,007347776
15	0,002088478	0,002088478	0,002088478	0,006265433
16	0,001783567	0,001783567	0,001783567	0,005350701
17	0,001530101	0,001530101	0,001530101	0,004590303
18	0,00132532	0,00132532	0,00132532	0,003975959
19	0,001171799	0,001171799	0,001171799	0,003515398
20	0,001098194	0,001098194	0,001098194	0,003294582

The variations of the expected gain are shown in Figure 3.8. The dash-dot line signifies the expected gain of the three jobs, and the solid line is the total of the expectations. The expected gain for the three jobs are the same, because the jobs have identical characteristics such as arrival rates, service rates, memory occupations.

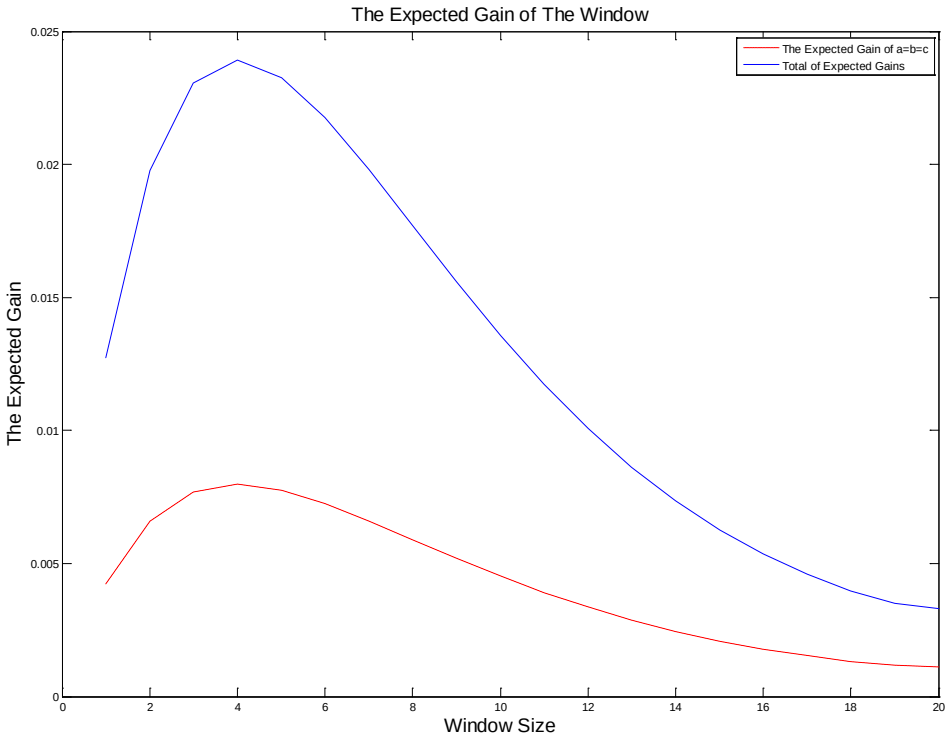


Figure 3.8 The Gain of the Window

An example is shown in Table 3.3. If the sliding window methodology was not used, the job “a” would occupy the memory for a long time. Even it is not needed anymore, because of it’s repetition count and it’s other selection criterion, “a” would rest in the memory. If no jobs are kept in the memory the system endures for 18 ms. But we save approximately 12 ms and the system endures only 6 ms. But in the example is shown in Table 3.3, the jobs are distributed individually and the gain is only 8 ms.

Table 3.3 Example 1

The jobs	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a
	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a
	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a
	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b
	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b
	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b
	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c
	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c
	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c
	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a
	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a
	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a
	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b
	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b
	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b
	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c
	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c
	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c
The Gain of The Window	0,004241	0,006589	0,007682	0,007969	0,007757	0,007969	0,007682	0,007969	0,007757	0,007969	0,007682	0,007969	0,007757	0,007969	0,007682	0,007969	0,007757	0,007969
Error	0,019667	0,017319	0,016225	0,015938	0,016151	0,015938	0,016225	0,015938	0,016151	0,015938	0,016225	0,015938	0,016151	0,015938	0,016225	0,015938	0,016151	0,015938
Memory	a	a	a	a	a	a	b	b	b	a	a	a	a	a	a	b	b	b
				b	b	b	c	c	c	c	c	c	b	b	b	c	c	c
Reel Gain		a exists	a exists		b exists	b exists		c exists	c exists		a exists	a exists		b exists	b exists		c exists	c exists
		1	1		1	1		1	1		1	1		1	1		1	1

For a job set as in Table 3.3, the algorithm starts with window size equals to one. The first job is “a”, and it is not in the memory. At the beginning, the memory is free, and the window size equals to one. The difference between the total expectation of the jobs and the expectation of job “a” equals to 0.019667 when the window size is one. It is smaller than the position at window size equals to zero, 0.023908. Because of the emptiness of the memory, means 2 bytes can be kept in the memory, and “a” has an output with size equals to 1. The output of the job “a” is kept in the memory. The algorithm decides to increase the window size.

The second job is again “a”. It’s output is already in the memory. The error equals to 0.017319 which is smaller than 0.019667 (previous error). The algorithm decides that it done well with previous iteration that is increasing window size, and continues to increase window size. After the third iteration the error becomes 0.016225 which is smaller than previous error. After fourth iteration the error becomes 0.015938. The fifth iteration is done by window size equals to five. But the error becomes 0.016151 which is greater than previous error. The algorithm decides to undo last window action and sets window size to 4, and continues to change the window size at each iteration. Because of the limit of the memory equals to 2 bytes and each output keeps one byte, only two outputs can be kept in the memory at same time. Each job endures one millisecond. At total 18 ms jobs has done, but it endured only 6 ms.

Table 3.4 Example 2

The jobs	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a
	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b
	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c
	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a
	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b
	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c
	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a
	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b
	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c
	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a
	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b
	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c
	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a
	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b
	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c
	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a
	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b
	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c
The Gain of The Window	0,004241	0,006589	0,007682	0,007969	0,007757	0,007969	0,007682	0,007969	0,007757	0,007682	0,007969	0,007757	0,007969	0,007682	0,007969	0,007757	0,007969	0,007682
Error	0,019667	0,017319	0,016225	0,015938	0,016151	0,015938	0,016225	0,015938	0,016151	0,016225	0,015938	0,016151	0,015938	0,016225	0,015938	0,016151	0,015938	0,016225
Memory	a	a	a	a	a	b	b	b	b	b	b	b	a	a	a	a	a	a
		b	b	b	b	c	c	c	c	c	c	c	c	c	c	c	b	b
Reel Gain				a exists	b exists			b exists	c exists		b exists	c exists			c exists	a exists		
				1	1			1	1		1	1			1	1		

If these jobs are arranged in an order as Table 3.4, the real gain is only 8 ms. Because the jobs whose outputs are kept in the memory are recalled after the other jobs called. The jobs in the window do not mean that the outputs of all of them are kept in the memory, it means they can be kept in the memory if there exist enough place.

3.5 The Desired Value Derivation

Our goal is to minimize the service time of the system. In other words, the time cost of obtaining the outputs of the jobs has to be minimized. Thus, by their first process, all jobs have to exist in the memory. If we could have an unlimited memory, we would have kept all jobs in the memory, but unfortunately despite of the development of hardware systems we have limited system resources.

N_{Total} is called as number of jobs processed and it is bigger than Ω number of jobs in a window. T_{Total} is the total service time elapsed for all the processed jobs having the subindices by k. Equation (3.20) can be obtained;

$$T_{Total} \simeq \frac{\sum_{k=1}^{N_{Total}} T_k(\alpha)}{\Omega} \tag{3.20}$$

Where $\alpha = \frac{m}{m_T}$ (proportional usage of the memory).

Minimizing T_{Total} is the desired value. The time cost of running the I/O jobs are not in the scope of this study (if there is not enough place in the memory to run new jobs, the time cost of running the I/O jobs will increase but a condition can be added to the system to cover this case). In conclusion, our goal is to obtain all computational jobs in the memory by their first call.

$$v_d = \sum_{i=1}^{m} t_r^i \tag{3.21}$$

A condition can be added to ensure that there exists still enough memory. If the running time of one consequent job is bigger than the average of the time cost of running jobs, it will start to organize jobs which are in the memory by deleting most inconvenient job in the memory. In other way, desired value can be described as formula (3.19) which equals to the expected value.

3.6 Error Derivation

Through the definition, the error is the difference between the obtained result and the desired value. In other words, error is described as;

$$e_k = \max_{\tau \in \mathbb{N}}(E(\Psi)) - \sum_{i=k-\tau+1}^{k} \frac{\Psi_i p_i}{\tau} \tag{3.22}$$

The effects of new job states on the error are shown in Table 3.5.

Table 3.5 The effects of the new job arrival and the error variation.

The status of the new job that is entering into the window	The status of the old job that is leaving the window	The effect on the error
Exists in the memory	Exists in the memory	Does not change
Exists in the memory	Does not exist in the memory, computational job	Decrease
Exists in the memory	Does not exist in the memory, I/O job	Decrease
Does not exist in the memory, computational job, terminated in average time	Exists in the memory	Does not change
Does not exist in the memory, computational job, terminated in average time	Does not exist in the memory, computational job	Does not change
Does not exist in the memory, computational job, terminated in average time	Does not exist in the memory, I/O job	Does not change
Does not exist in the memory, computational job, terminated in more than average time	Exists in the memory	Increase
Does not exist in the memory, computational job, terminated in more than average time	Does not exist in the memory, computational job	Does not change
Does not exist in the memory, computational job, terminated in more than average time	Does not exist in the memory, I/O job	Does not change

Does not exist in the memory, I/O job, terminated in average time	Exists in the memory	Does not change
Does not exist in the memory, I/O job, terminated in average time	Does not exist in the memory, computational job	Does not change
Does not exist in the memory, I/O job, terminated in average time	Does not exist in the memory, I/O job	Does not change
Does not exist in the memory, I/O job, terminated in more than average time	Exists in the memory	Increase
Does not exist in the memory, I/O job, terminated in more than average time	Does not exist in the memory, computational job	Does not change
Does not exist in the memory, I/O job, terminated in more than average time	Does not exist in the memory, I/O job	Does not change

The decisions on the situation of the memory can be summarized as given in Table 3.4. In the situations marked by “decreasing” change, the service time starts to decrease and the computational jobs can be kept in the memory. At “no change” points, the duration decrease continues or the stationary duration exists or the system is in the duration before the increment. Because of these circumstances cannot be separated, keeping new jobs in the memory can continue. At the “increasing” points, the time cost of the jobs which started to be processed becomes excessive. In this situation, the jobs in the memory should be deleted.

At this point to decide which job will be deleted from the memory is essential for the system. Figure 3.9 shows the decision automata which describes decision states of the system.

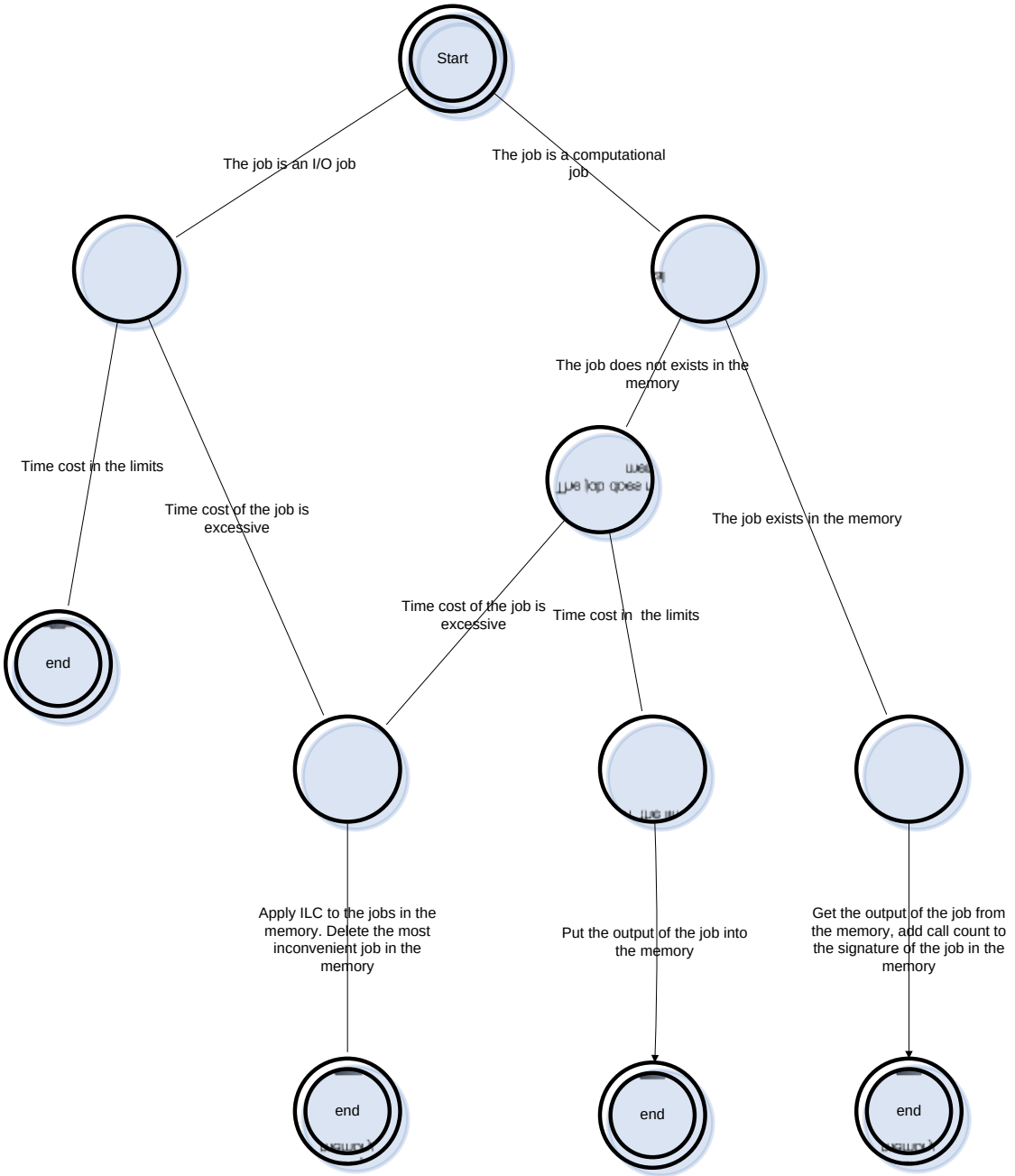


Figure 3.9 The decision automata

The generic code for the decision algorithm is given below;

```
for each job begin
    if the job is an I/O job begin
        t = calculate_time_cost(run(job), output)
        if t < time_limit_for_I/O_job begin
            recalculate(time_limit_for_I/O_job,t);
            return output;
        end;
        else begin
            ILC(job_list_in_the_memory)
            delete_most_inconvenient_job(job_list_in_the_memory)
        end;
    end;
    else begin
        if is_in_the_list(job, job_list_in_the_memory)begin
            increase_call_count(job, job_list_in_the_memory);
            return get_from_memory(job, job_list_in_the_memory);
        end;
        else begin
            t = calculate_time_cost(run(job),output)
            if t < time_limit_for_Computational_job begin
                recalculate(time_limit_for_Computational_job, t);
                add_job(job, job_list_int_the_memory);
                return output;
            end;
            else begin
                ILC(job_list_in_the_memory);
                delete_most_inconvenient_job(job_list_in_the_memory);
                return output;
            end;
        end;
    end;    end;
```

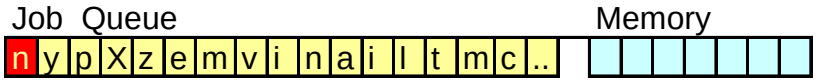
3.7. Processing Jobs

An example of the system is specified at Figure 3.10. Each character signifies unique job, the “X” signify an I/O job. The system is composed by the job list, first memory to keep the outputs of the job and the window memory to keep the output of jobs in the window whose output does not exist in the first memory.

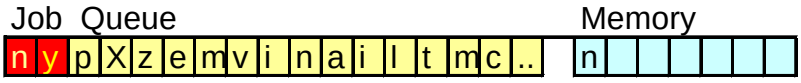
When the system starts to operate, the sliding window begins to change as Figure 3.10.a. The first window represented with red includes only first job. The memory that is used for keeping outputs of the jobs is empty. After processing the first job the window gets the second job, and the output of the first job is kept in the memory as Figure 3.10.b. The X cannot be set in the memory because it is an I/O job. The 6^{th} position of the window is shown in Figure 3.10.c. All precedence outputs are in the memory, and still there exists memory. The 10^{th} job is the same as the first job and the output of the 1^{st} job is still in the memory, it can be used without computing by the CPU meanwhile adding the job count to the object in the memory. “n” object has its call orders, 1 and 10 which will be used for calculating gain function for the job.

Now the limit of the memory is reached, but the system does not know that. When the 11^{th} job arrives, it is processed in excessive time because there is not enough memory to perform that job easily, and the system should choose one of the eight jobs to delete from memory. A gain function can be used to decide job to delete. The gain of “z” is the minimum value of all then the output of the “z” is deleted from the memory and the output of “a” is replaced.

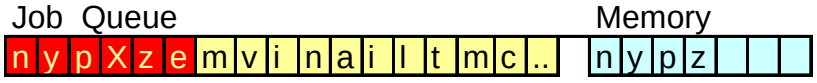
When the other jobs of the system require more memory, the size of the memory that is used for keeping the outputs, decreases by the time average condition of the works.



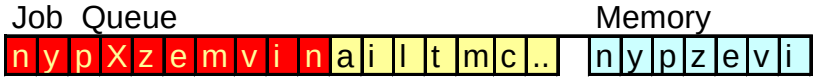
a) The first position of the window



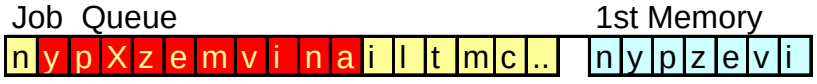
b) The second position of the window



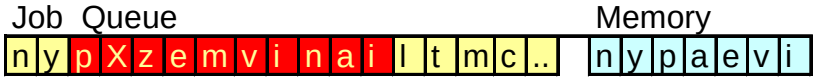
c) The 6th position of the window



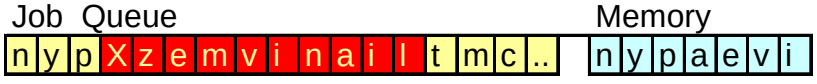
d) 10th position of the window



e) 11th position of the window



f) 12th position of the window



g) The 13th position of the window

Figure 3.10 Example of the flow

4. Dynamic Resource Allocation and Memory Scheduling

Iterative Learning Control methodology is used in our algorithm because previous experiences will be used for next memory scheduling. The goal is to minimize the error and to regulate the window size to ensure caching the most frequent repeating jobs' results in the memory to avoid unnecessary computing effort.

4.1 Iterative Learning Control

Iterative Learning Control is recognized as a powerful control method to achieve trajectory tracking tasks in a class of repetitive systems [11]. Our goal is to obtain the ideal window size such that the frequently repeated similar job results may be cached and to achieve best utilization of the processor. To achieve this goal, the window dimension has to be adjusted after the each iteration.

A typical Iterative Learning Algorithm control output formulation is given by, [19].

$$u_{k+1}(t) = u_k(t) + \gamma \frac{d}{dt} e_k(t) \tag{4.1}$$

Where $e_k(t)$ can be defined as the error term expressed as a difference between the desired expected gain of the overall jobs and the window gain.

$$e_k(t) = v_d(t) - v_r(t) = \max_{\tau \in \mathbb{N}} (E(\Psi)) - \sum_{i=k-\tau+1}^{k} \frac{\Psi_i p_i}{\tau} \tag{4.2}$$

The desired expected gain, which is the first term in the error function given by (4.2), is constituted by summing all the individual expected gain of the jobs existing in the memory (see also (3.19) for its derived expression). The second term is the averaged sum of the individual gain existing in the sliding window.

In our setting, it can be implemented as (4.3).

$$\tau_{k+1} = \tau_k - sign(\frac{e_k - e_{k-1}}{\tau_k - \tau_{k-1}}) \tag{4.3}$$

Where sign function is described as below,

$$sign(.) = \begin{cases} sign(x)=1 \text{ if } x \geq 0 \\ sign(x)=-1 \text{ if } x<0 \end{cases} \tag{4.4}$$

By the way the system tries to reach the optimum size of the window dynamically and it is temporal variable. The CPU usage derivations of Example 1 and Example 2 can be seen in Figure 4.1.

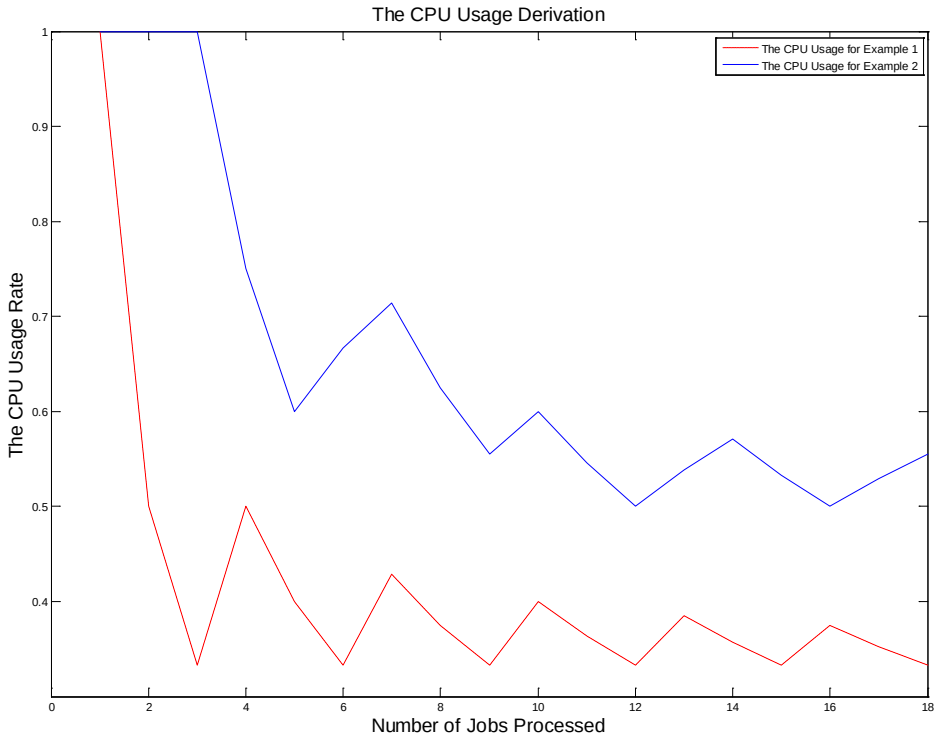


Figure 4.1 Error Derivation

In the Figure 4.1 the CPU usage rate decreases exponentially and exhibiting some oscillations due to the discontinuous nature of the window size regulation which means more outputs are found in the memory. At the beginning, the first job (job “a”) is processed by the CPU for both examples. In Example 1, described as dash-dot line, the second jobs (job “a”) output is found in the memory; it is not processed by the CPU, the usage decreased to half of its initial value. The third job is again “a” and the usage decreases to its thirty percent. In Example 2, described as solid line, the second and the third jobs (job “b” and job “c”) are processed by CPU. The forth job is “b” in Example 1. It’s output does not exist in the memory. The CPU usage becomes half, because the first “a” and “b” could not been found in the memory and they were processed but the second and the third “a” were found in the memory.

A simulator has been prepared to observe gains with different types of random systems. Different number of jobs was chosen to generate a system with 10000 jobs. The service

rates were chosen between 1 and 10 and the footprints of outputs were chosen between 1 and 100, both with uniform distribution. The jobs sequence was chosen with uniform distribution.

Figure 4.2 shows the variation of Expected Values that has a maximum value when window size equals to 187, and Figure 4.3 shows the variation of Error and Figure 4.4 shows the derivation of CPU usage with 50 jobs and capacity equals to 200. Figure 4.5 shows the derivation of window size, at the beginning there one job and the window size is started with one, and increased by one by one until error value decreases. It can be observed from Figure 4.3 and 4.5, the error derivations and the window size derivations are almost contrary.

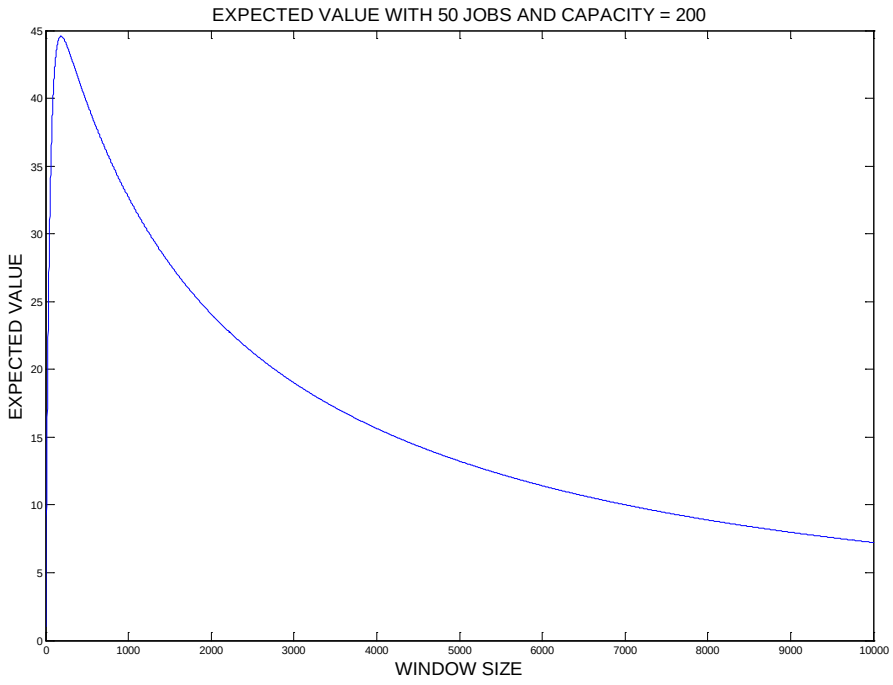


Figure 4.2 Expected Value with 50 jobs and capacity = 200

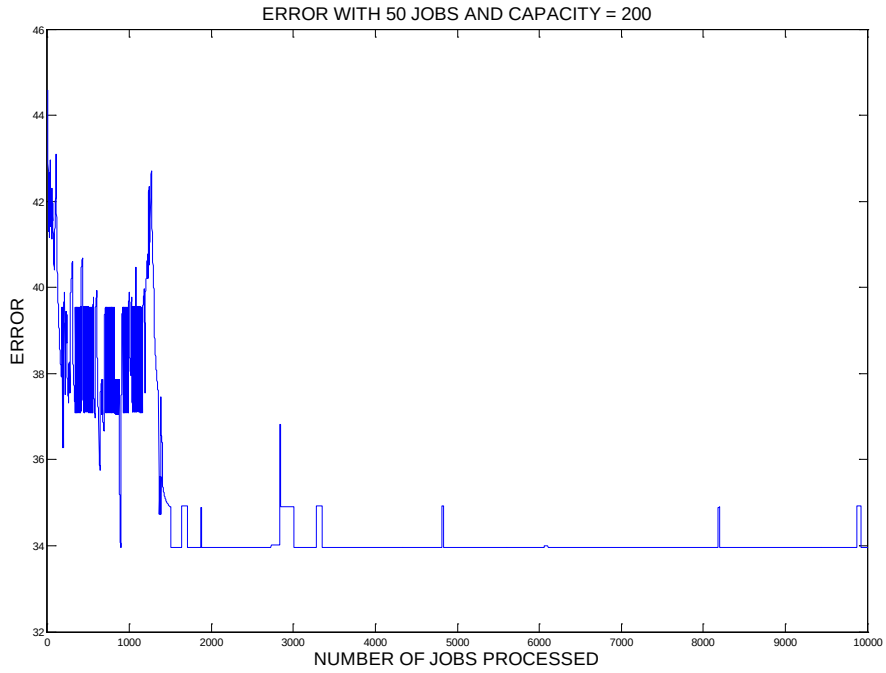


Figure 4.3 Error variation with 50 jobs and capacity = 200

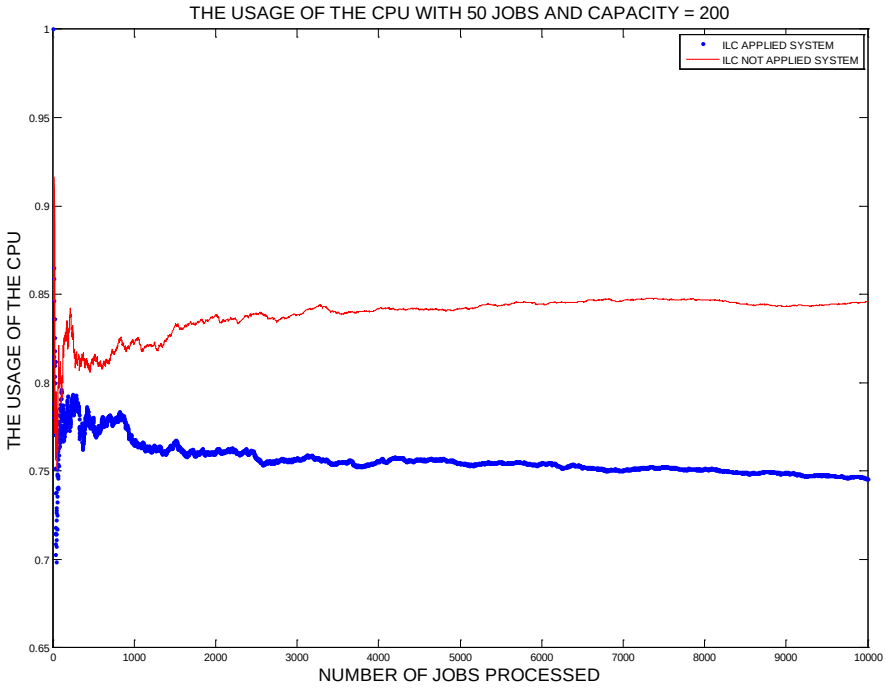


Figure 4.4 The Usage of the CPU with 50 jobs and capacity = 200

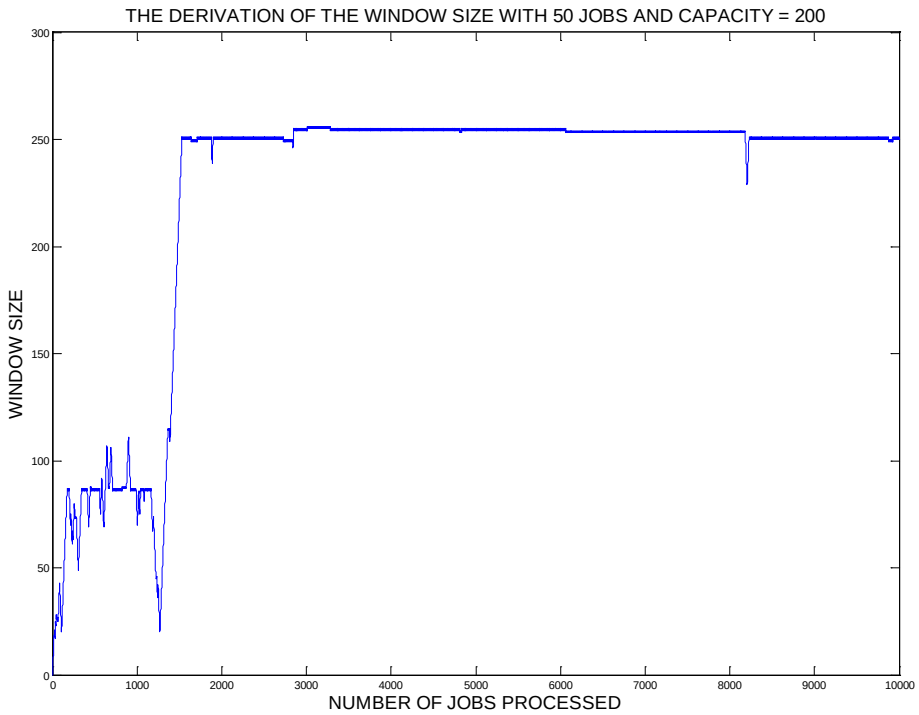


Figure 4.5 The Derivation of the window size with 50 jobs and capacity = 200

The footprints are distributed between 1 and 100 with uniform distribution. The average of service rates is about 50 (In this example the sum of the footprints are 2347). About 4 jobs can be kept in the memory with capacity 200. But Figure 4.4 shows that when the system goes to stationary state, 25% of jobs are not reprocessed, but if Iterative Learning Control Methodology is not used 15% of jobs are not reprocessed.

Figure 4.6 shows that the variation of the Expected Values has maximum value when window size equals to 168, and Figure 4.7 shows the variation of Error and Figure 4.8 shows the derivation of CPU usage with 40 jobs and capacity equals to 400. Figure 4.9 shows the derivation of the window size with 40 jobs and capacity equals 400 system.

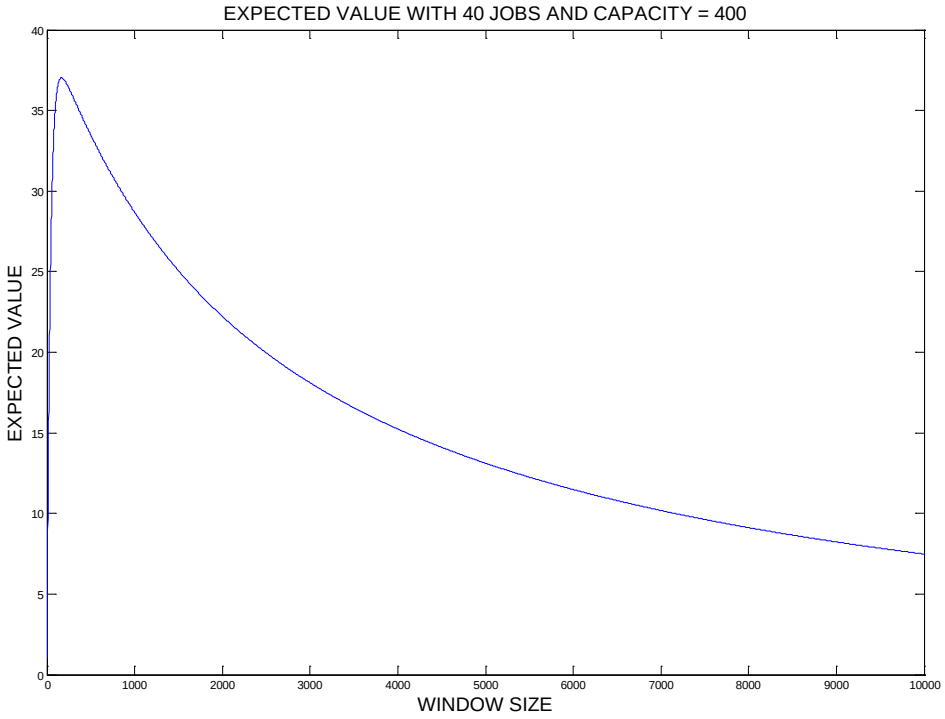


Figure 4.6 Expected Value with 40 jobs and capacity = 400

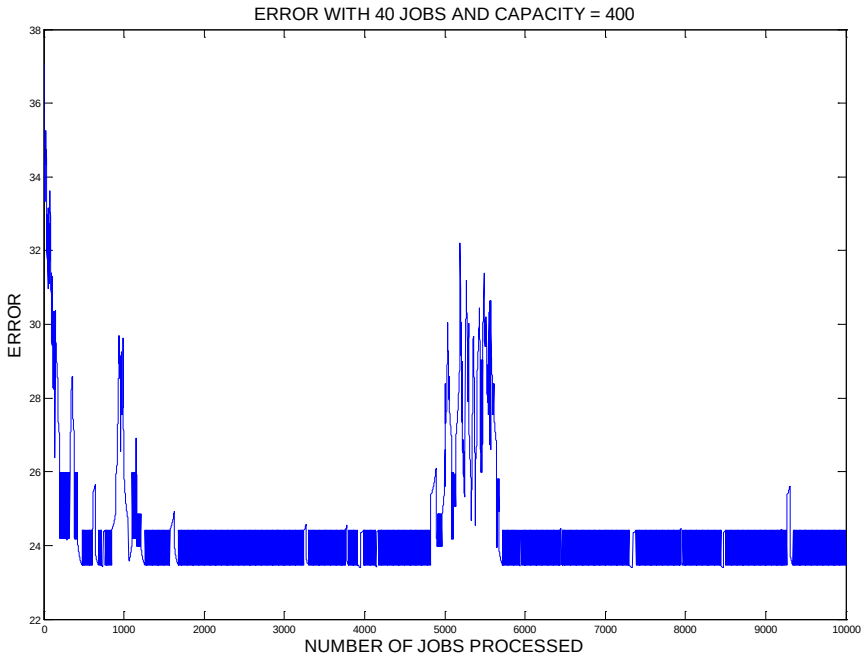


Figure 4.7 Error variation with 40 jobs and capacity = 400

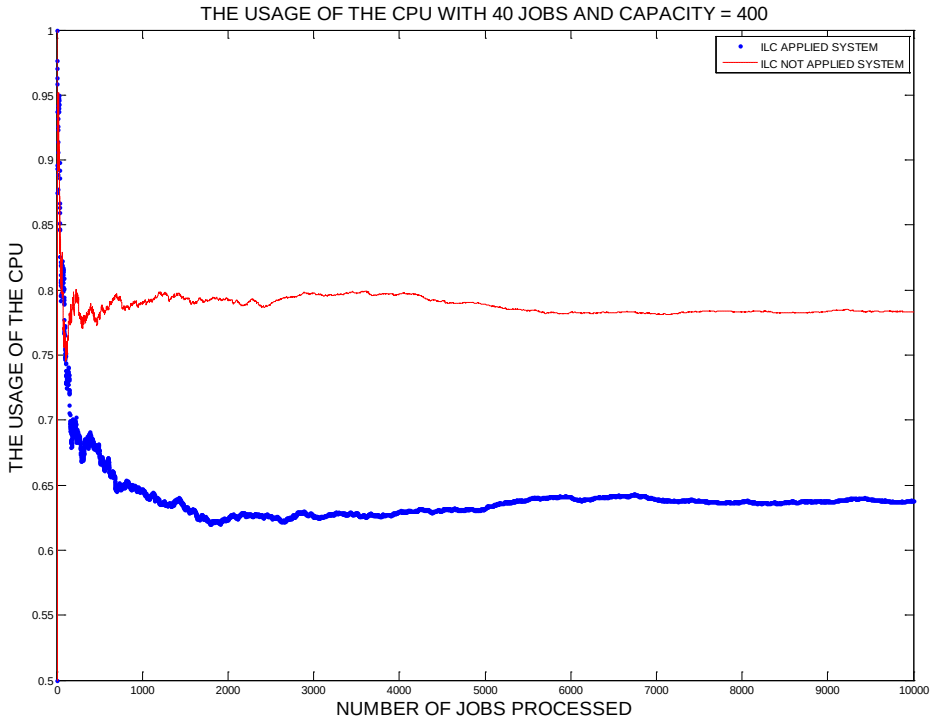


Figure 4.8 Relative gain with 40 jobs and capacity = 400

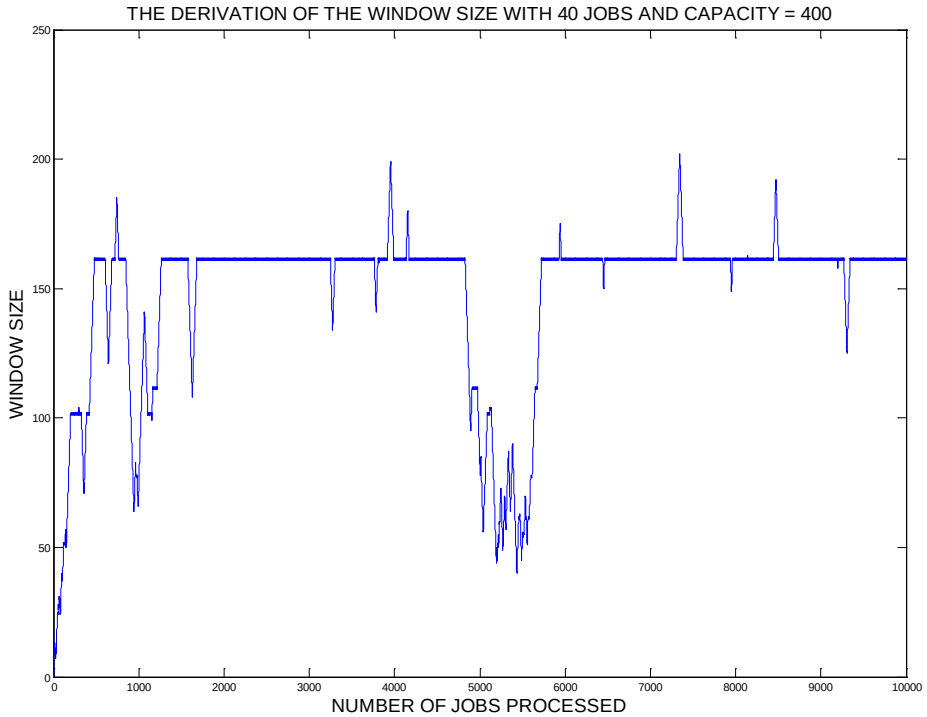


Figure 4.9 Derivation of the window size with 40 jobs and capacity = 400

About 8 jobs can be kept in the memory with capacity 400. But Figure 4.8 shows that when the system goes to stationary state, 35% of jobs are not reprocessed, but if Iterative Learning Control Methodology is not used 20% of jobs are not reprocessed. Between 4800 and 5800 the error rate is increasing in Figure 4.7. When the job order is analyzed, it is observed that a job set that does not exist in the memory arrives to the CPU.

Figure 4.10 shows the variation of Expected Values, and Figure 4.11 shows the variation of Error and Figure 4.12 shows the derivation of CPU usage with 100 jobs and capacity equals 1400 system. Figure 4.13 shows the derivation of the window size with 100 jobs and capacity equals 1400 system.

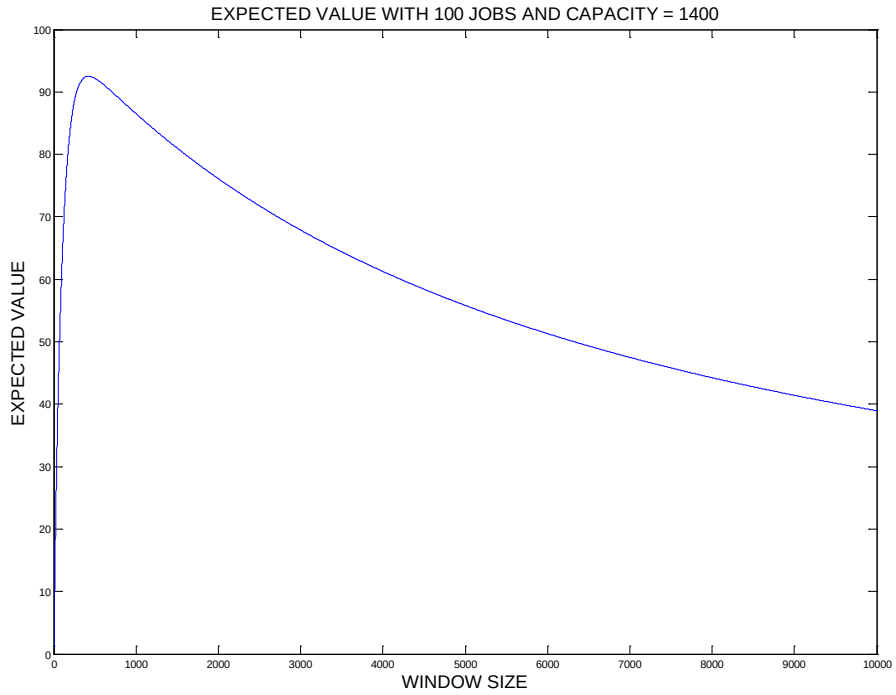


Figure 4.10 Expected Value with 100 jobs and capacity = 1400

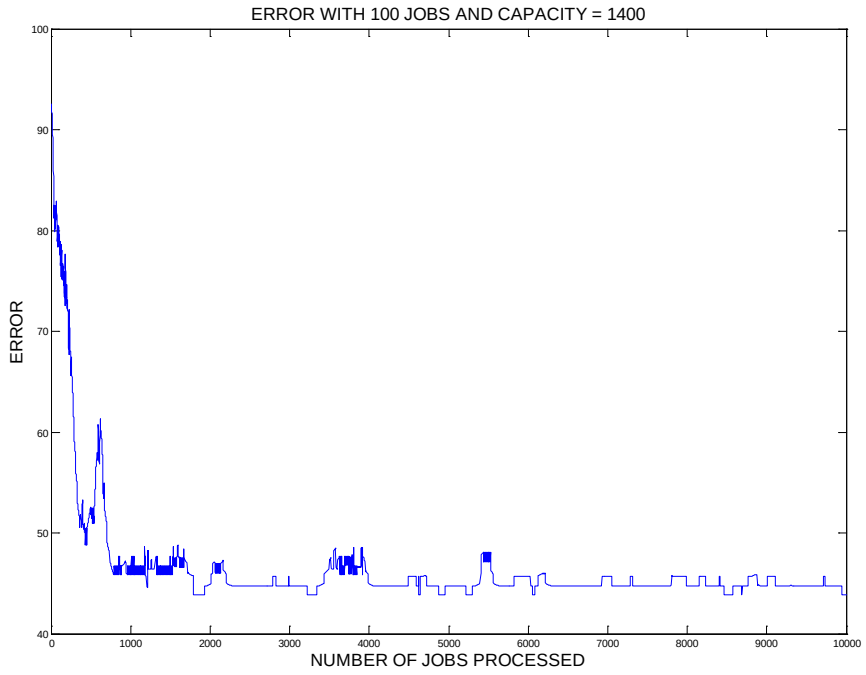


Figure 4.11 Error variation with 100 jobs and capacity = 1400

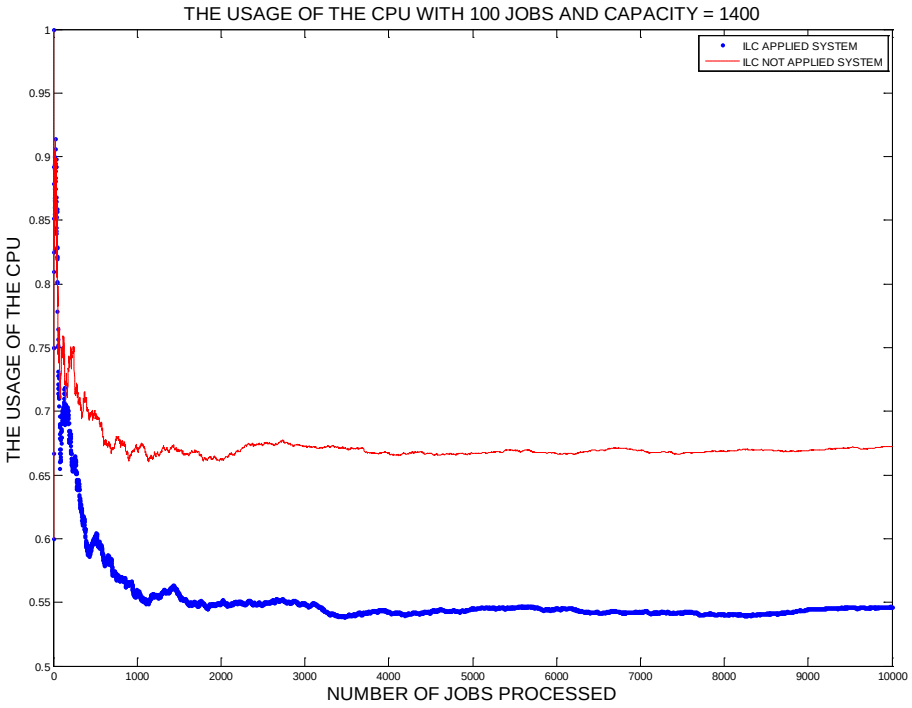


Figure 4.12 Relative gain with 100 jobs and capacity = 1400

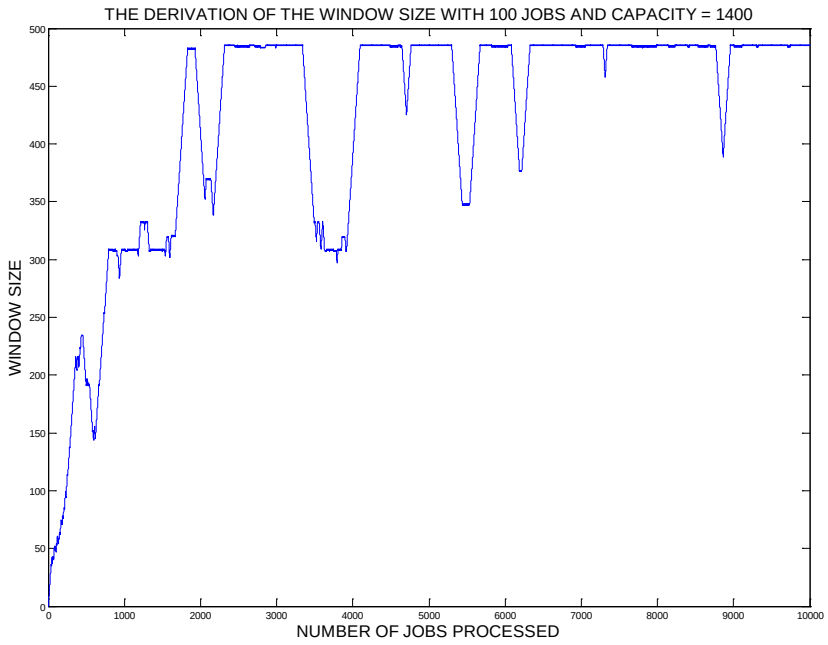


Figure 4.13 Derivation of the window size with 100 jobs and capacity = 1400

About 28 jobs can be kept in the memory with capacity 1400. But Figure 4.12 shows that when the system goes to stationary state 33% of jobs are not reprocessed, but if Iterative Learning Control Methodology is not used 46% of jobs are not reprocessed.

Figure 4.14 shows the variation of Expected Values, and Figure 4.15 shows the variation of Error and Figure 4.16 shows the variations of relative gain of 100 jobs and capacity equals 8000 system. Figure 4.17 shows the derivation of the window size with 100 jobs and capacity equals 8000 system.

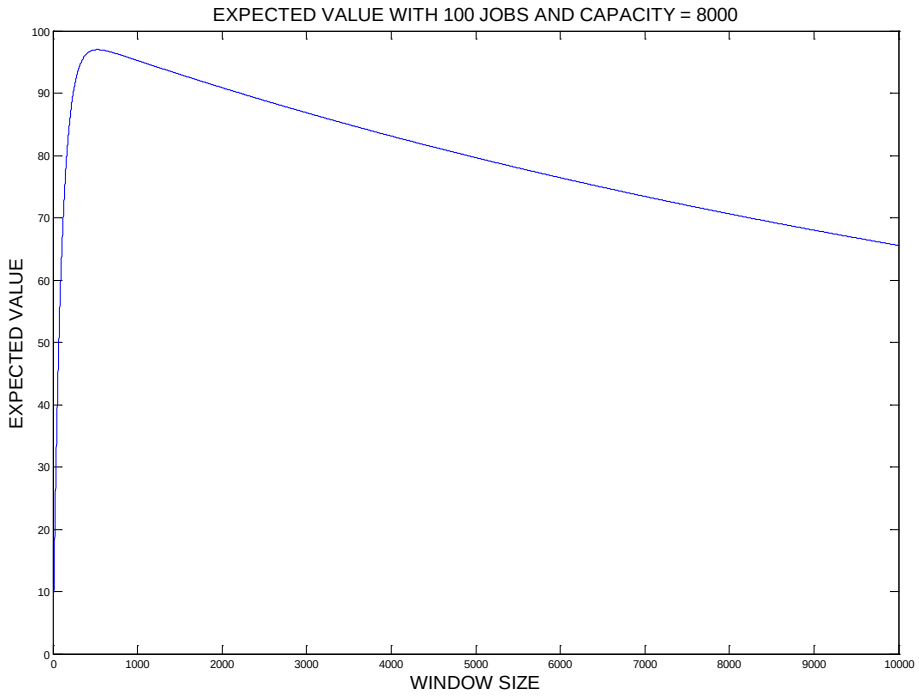


Figure 4.14 Expected Value with 100 jobs and capacity = 8000

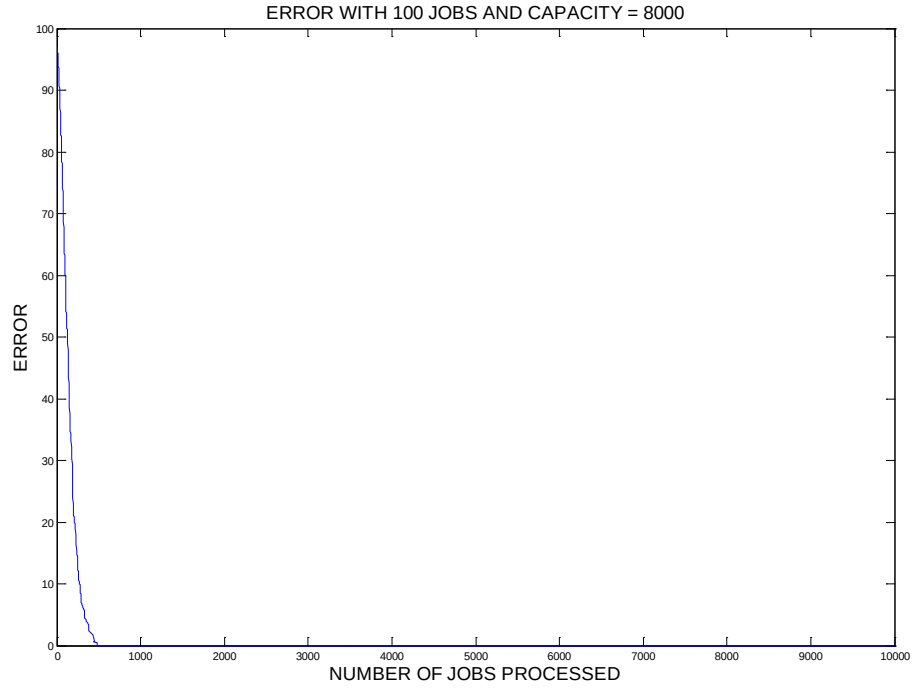


Figure 4.15 Error variation with 100 jobs and capacity = 8000

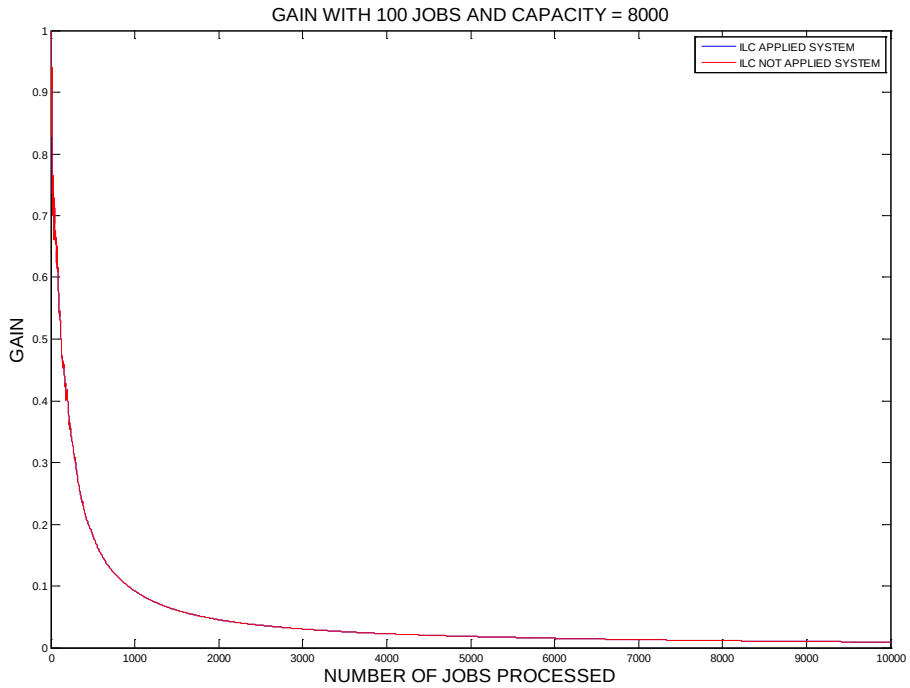


Figure 4.16 Relative gain with 100 jobs and capacity = 8000

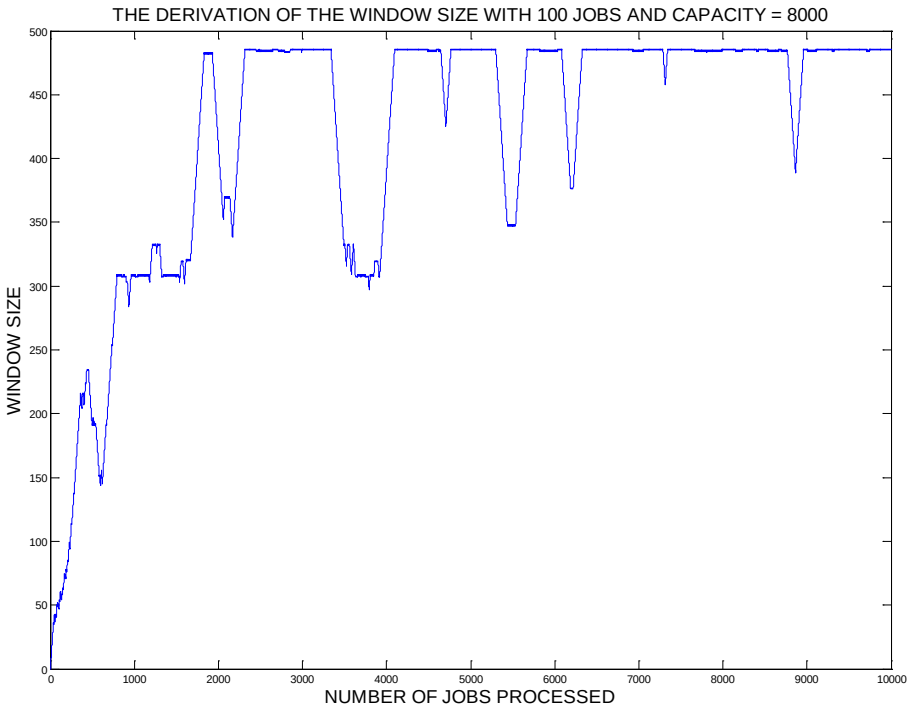


Figure 4.17 Derivation of the window size with 100 jobs and capacity = 8000

About 160 jobs can be kept in the memory with capacity 8000, but in these example total footprints is 4994. All of the jobs' outputs can be kept in the memory. It can be seen in Figure 4.16 that applying Iterative Learning Control methodology does not provide any benefit when the system has a huge memory. But when the memory is limited (which means real world case), Iterative Learning Control Methodology provides benefits. In the Figure 4.17, there exist window size changes because the arriving jobs are changing and the expected values are changed.

When Iterative Learning Control is applied CPU Usage Derivations are shown in Figure 4.18 and when Iterative Learning Control is not applied CPU Usage Derivations are shown in Figure 4.19. The difference of two case shows that if the memory capacity is limited, the gain of using Iterative Learning Control is higher as shown in Figure 4.20. At some points the differences are not high, there exists valleys in Figure 4.20. The arriving order of the jobs causes these cases.

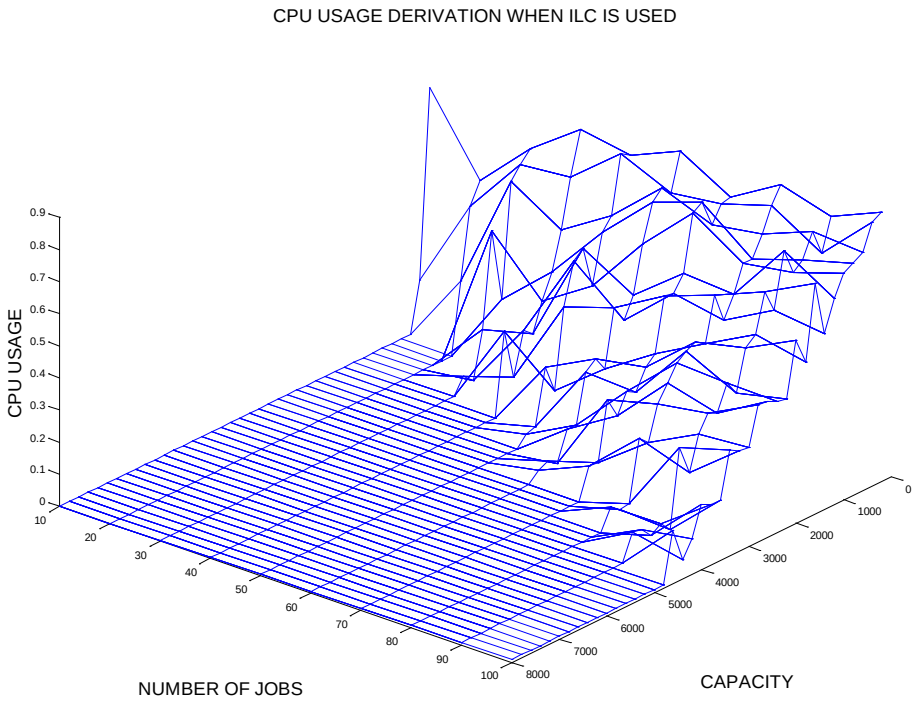


Figure 4.18 CPU Usage Derivations When ILC Applied

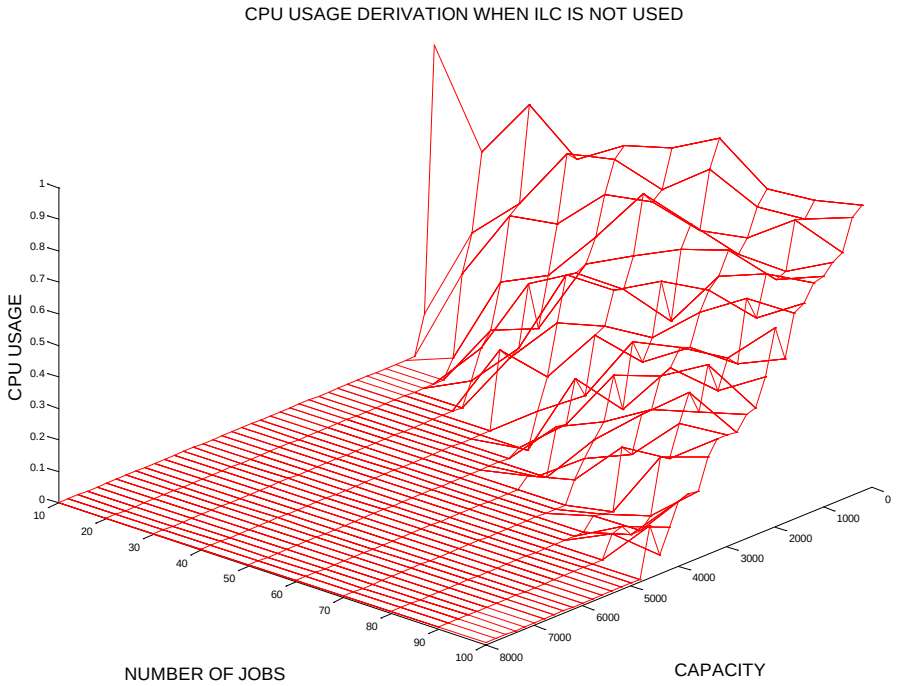


Figure 4.19 CPU Usage Derivations When ILC Not Applied

Figure 4.20 CPU Usage Derivations differences when applying Iterative Learning Control and not applying Iterative Learning Control

5. Experimental Study

The data is taken from real-life data belonging to a Time-Machine application of an individual retirement and life insurance company. Because of privacy policies the real data is corrupted by a linear function. The retirement retention and insurance premium calculations are using CPU much, and using the tables which are permanent but voluminous.

5.1 Experimental Results

As shown in Figure 3.3 and Figure 5.1, the time graphics of windows of N jobs depends on α which creates a view of a valley. Our goal is to allocate the resources of the computing system at the bottom of this valley as much as possible. In general, the system starts to go down in the graphic when jobs start to keep in the memory. But afterwards the system starts to go up on the right side of the valley because of the time cost of un-cached jobs. The algorithm has to determine that change. But it can be seen easily that there exists local minimum point in the system, because the jobs are not homogeneous. If the raise is small then it can be supposed that the value is negligible.

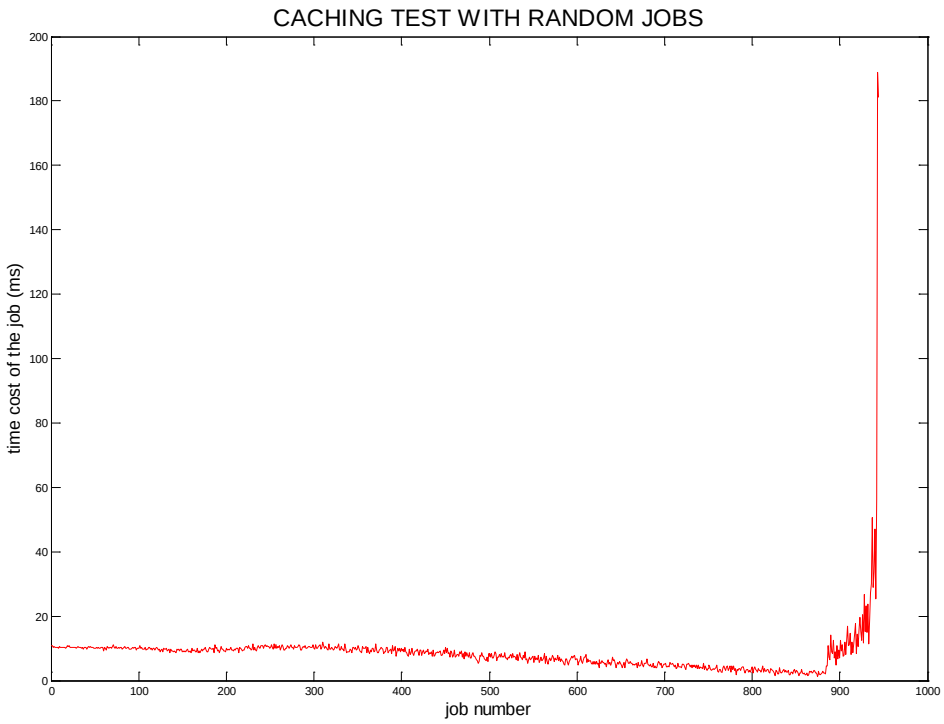


Figure 5.1 Caching test results with random jobs

The problem is which criteria will be utilized by the Iterative Learning Control system. Which job will be deleted from the memory in a situation that the jobs in the memory cause other jobs to run slowly? At that point a gain function is needed to compare the benefits of the jobs which are in the memory. The gain function has to be applicable to all computational jobs. The gain function has a direct proportion with the position of the job in the job list and it has a direct proportion with call frequency of the job and it has a direct proportion with the time cost of the job and it has an inverse proportion with the footprint of the output of the job in the memory. And the gain function is describes as g as below;

In this study, Java is used as programming language and Aspect Oriented Programming is used to implement our case study. Aspect Oriented Programming allows the separation of the functional mechanism from the non-functional ones [20]. In our study, an enterprise application (Retirement Projection Time Machine) is used and the outputs of the java methods are kept in the memory by Aspect Oriented Programming.

Retirement Projection Time Machine is a Java application that calculates future income of current payments and calculates required payments for desired income. The application also calculates life insurance premiums. The application is used by the customers and financial advisors of retirement and life insurance enterprises. The application runs on Websphere Application Server with heap size 256 Mb, 1.8 GHz Pentium Dual CPU and uses DB2 as Database. The application was developed by Struts framework as Model-View-Controller, and Hibernate was used for table-object modeling.

In the Figure 5.2 to 5.11 green lines represent the error graph of the desired value. If all jobs could be kept in the memory, the error would have been like that graph. The red lines represent the situation without dynamic control, until the system limit all jobs are kept in the memory. The blue lines represent the error value with the dynamic scheduling. The gain changes with different values of τ .

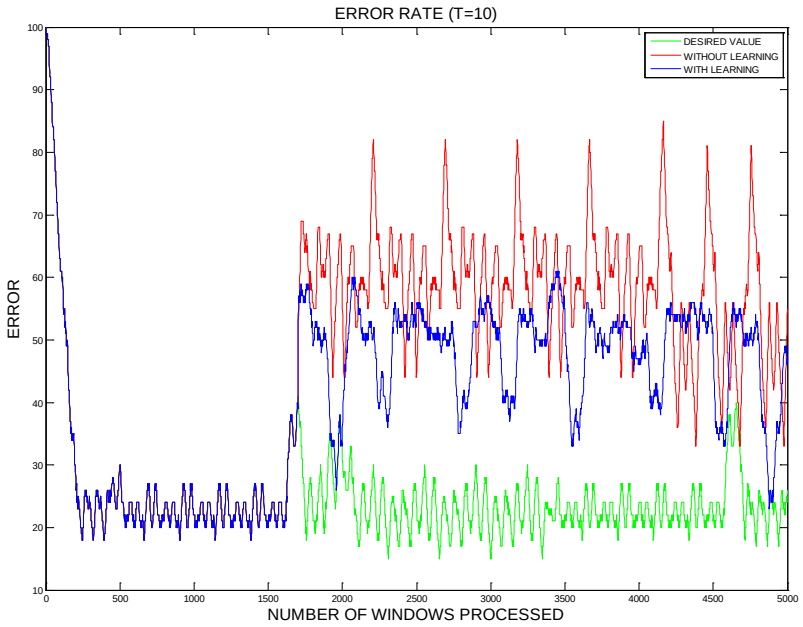


Figure 5.2 The Error Rate where τ =10

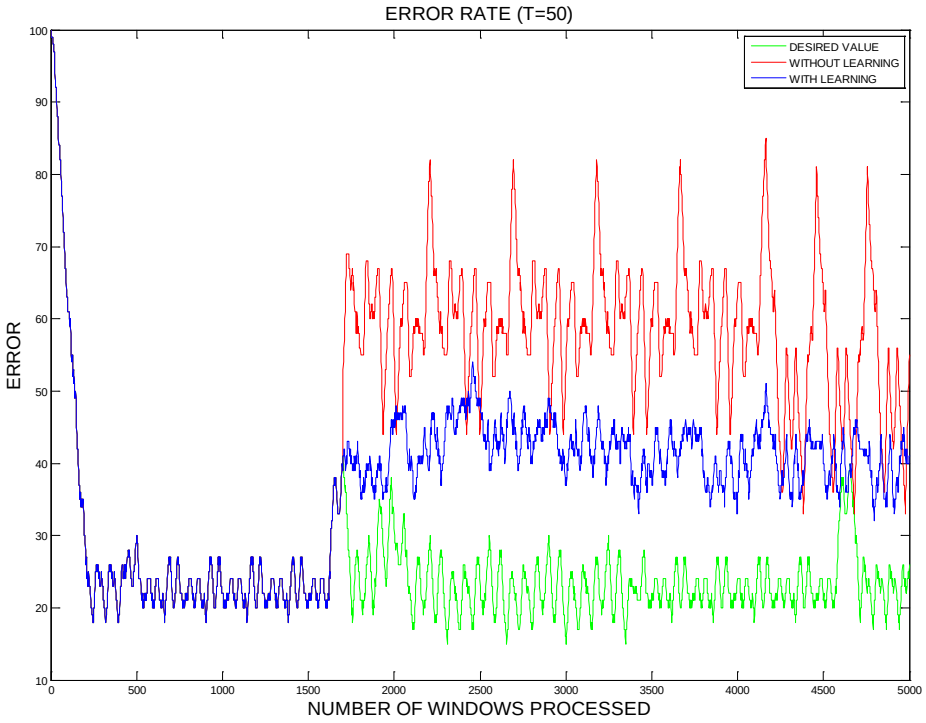


Figure 5.3 The Error Rate where τ =50

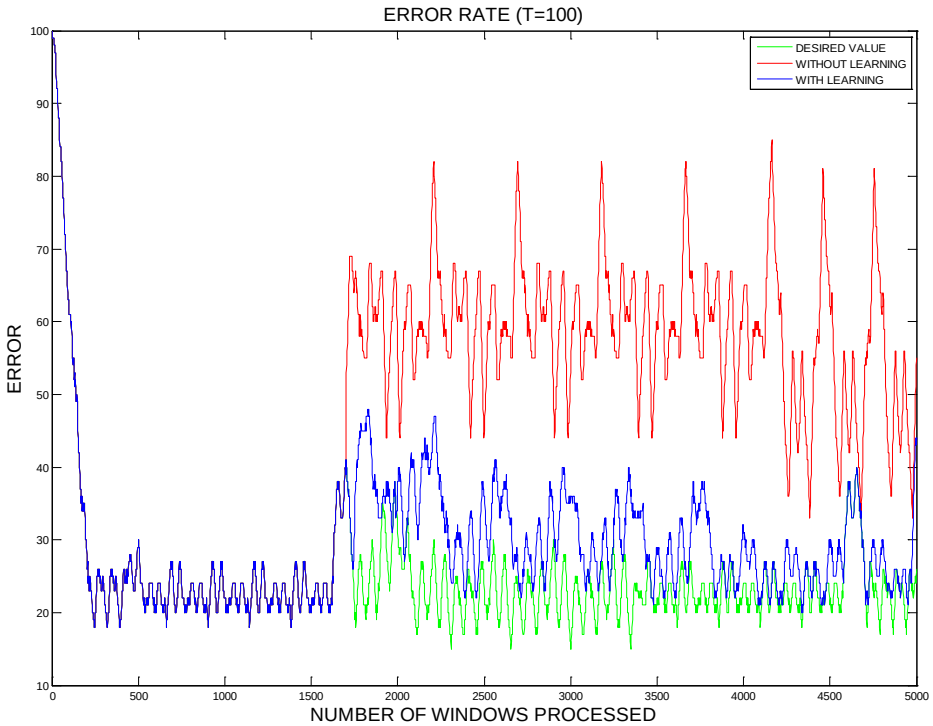


Figure 5.4 The Error Rate where τ =100

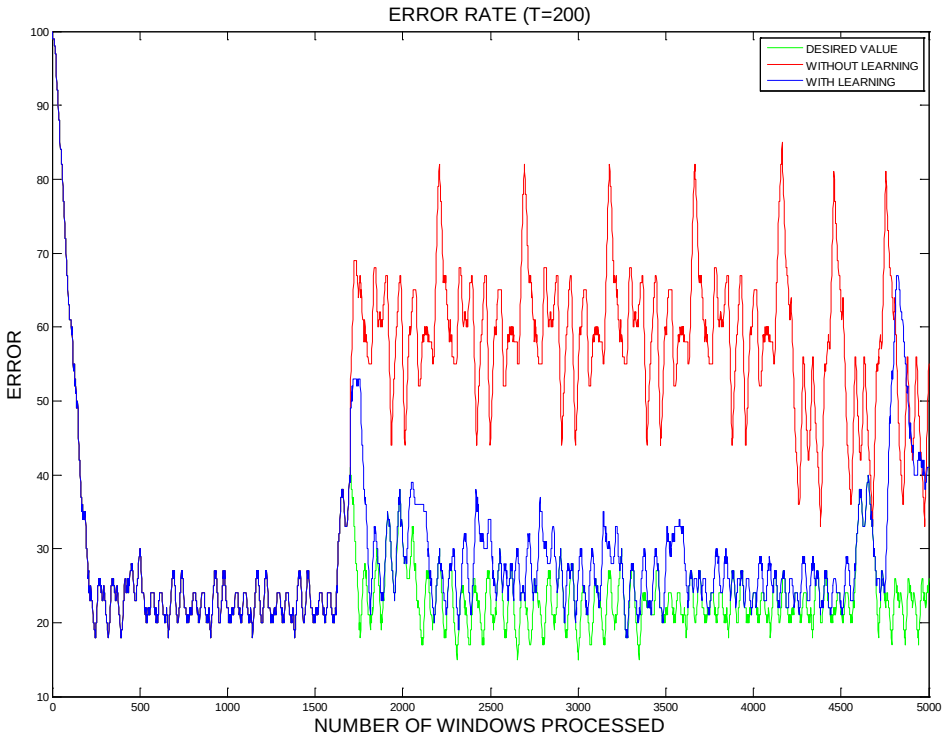


Figure 5.5 The Error Rate where τ =200

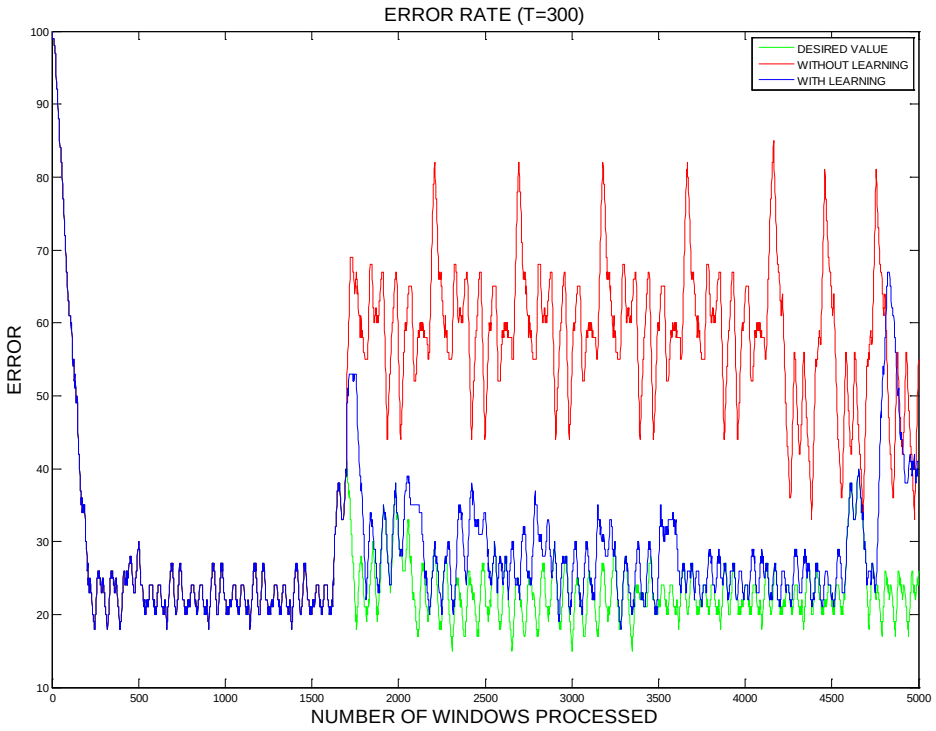


Figure 5.6 The Error Rate where τ =300

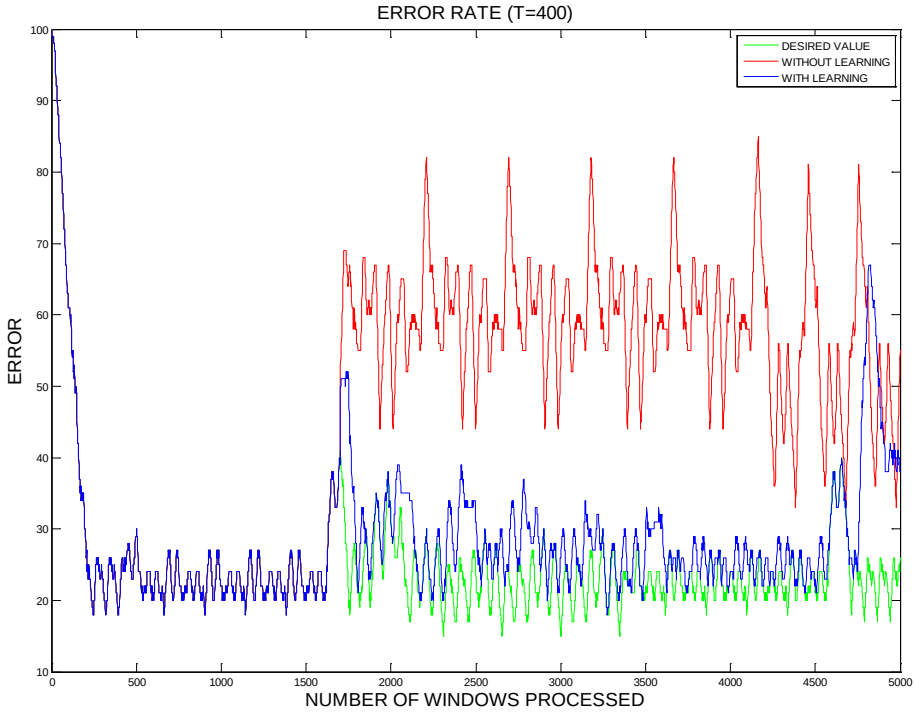


Figure 5.7 The Error Rate where τ =400

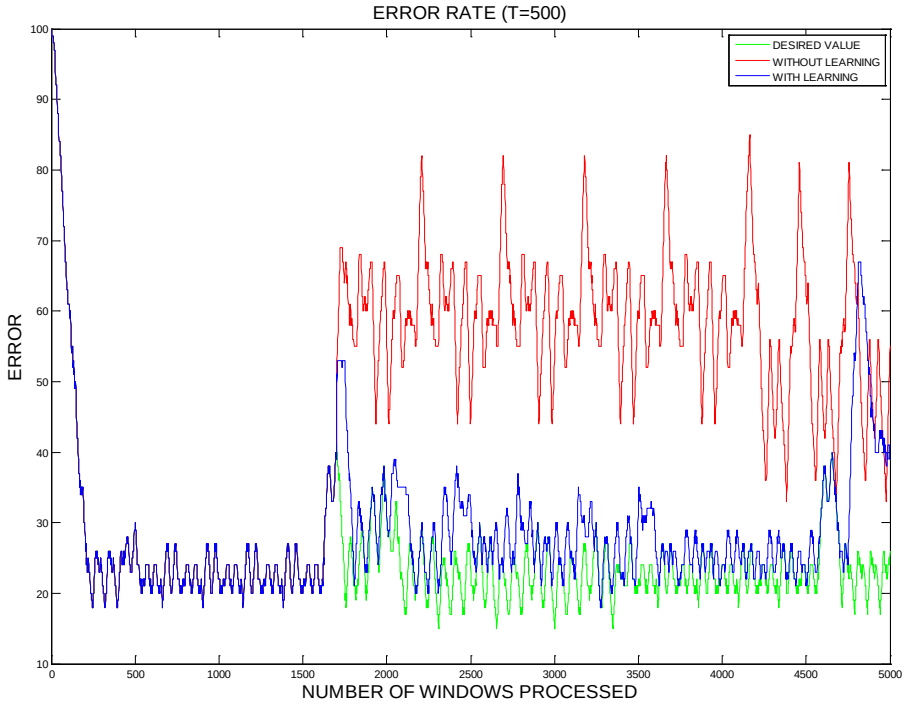


Figure 5.8 The Error Rate where τ =500

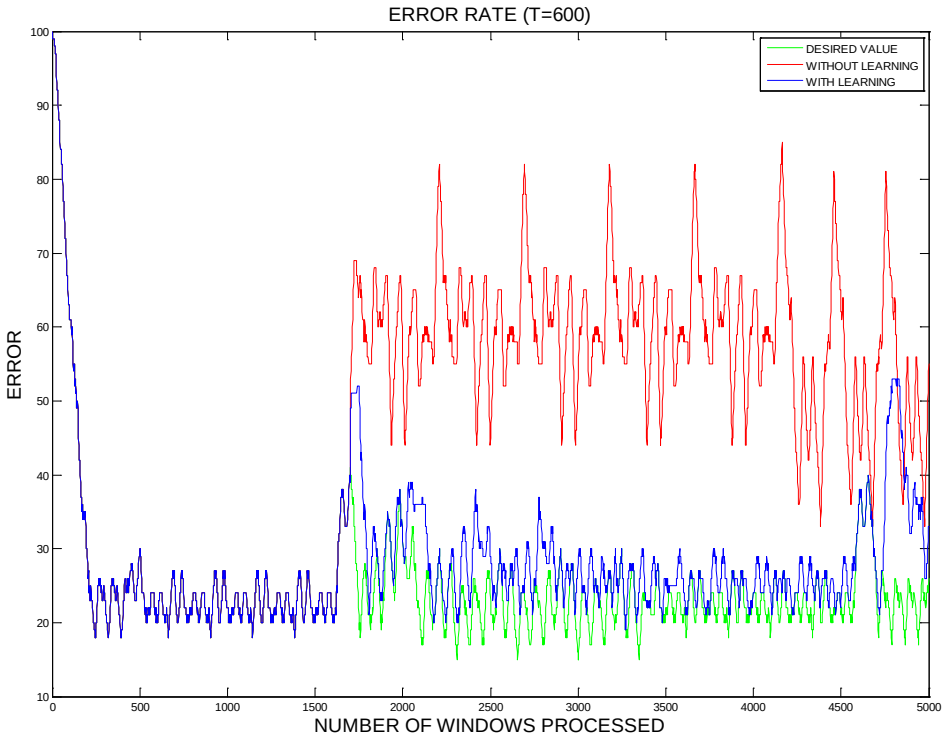


Figure 5.9 The Error Rate where τ =600

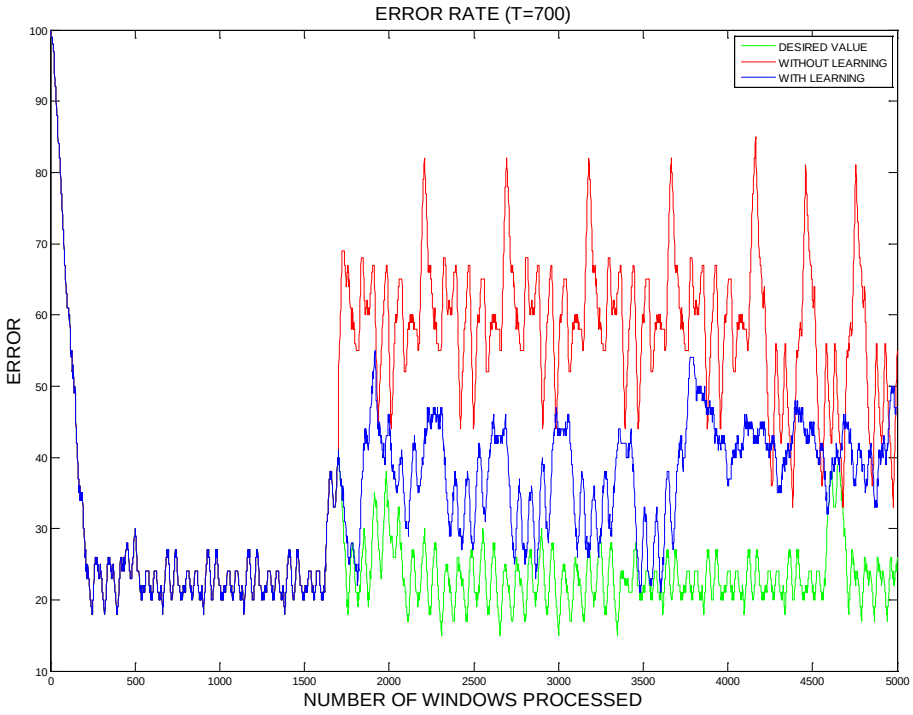


Figure 5.10 The Error Rate where τ =700

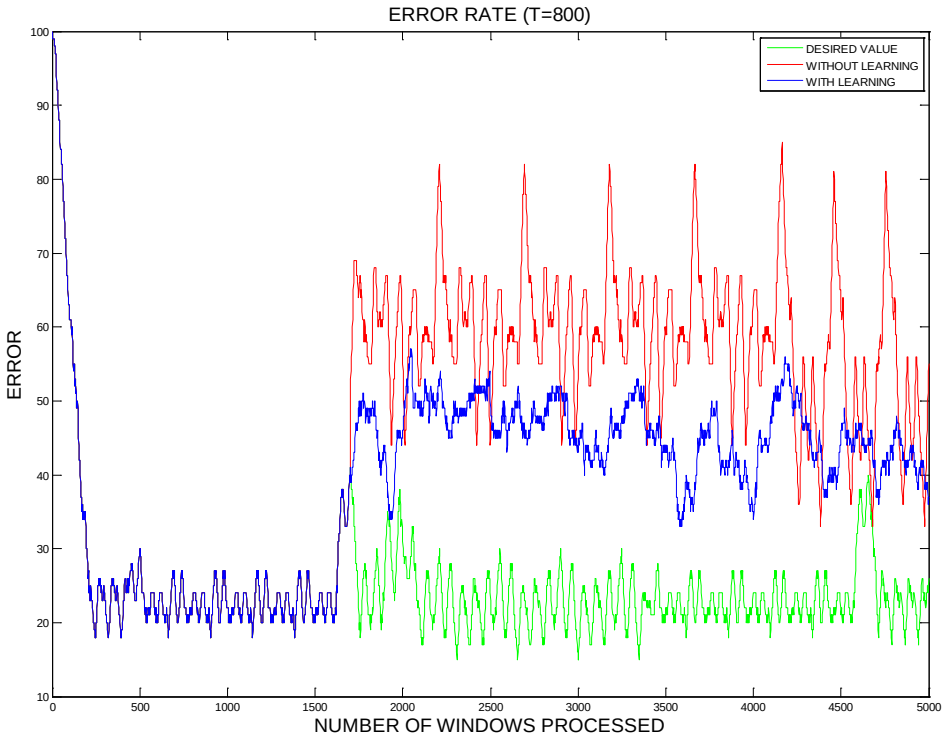


Figure 5.11 The Error Rate where τ =800

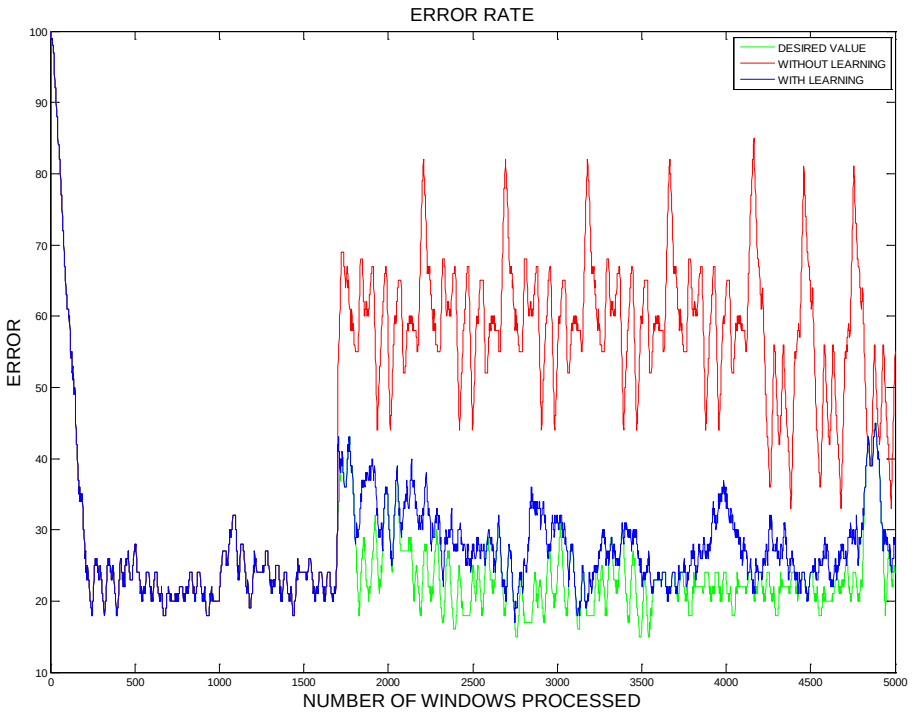


Figure 5.12 The Error Rate where τ calculated dynamically

Figure 5.12 shows the error rate when τ is calculated by Iterative Learning Control. The Figure 5.12 indicates that by this work we approached the goal that to enhance processing performance by dynamic memory scheduling.

In the previous experiences, all of the jobs characteristics are supposed to be known. But in real-life, it is impossible. To overcome this situation, the system is designed as the averages of previous weeks data is based on. When a new job arrives, it is statistical values are applied to next day's work. In Figure 4.13, it is shown that the CPU Usage of the first day after a week. It can be observed in Figure 4.13, Figure 4.14 and 4.15, the usage of CPU is dependant from the jobs arrive order, and each day has its own characteristic.

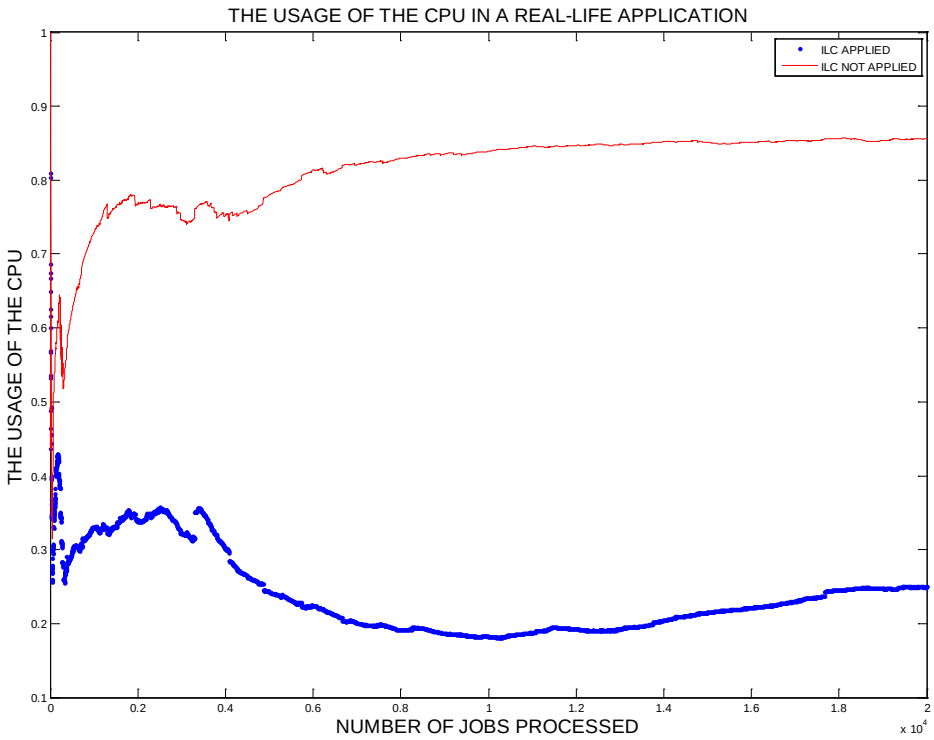


Figure 5.13 The Usage of CPU in a real-life application (First Day)

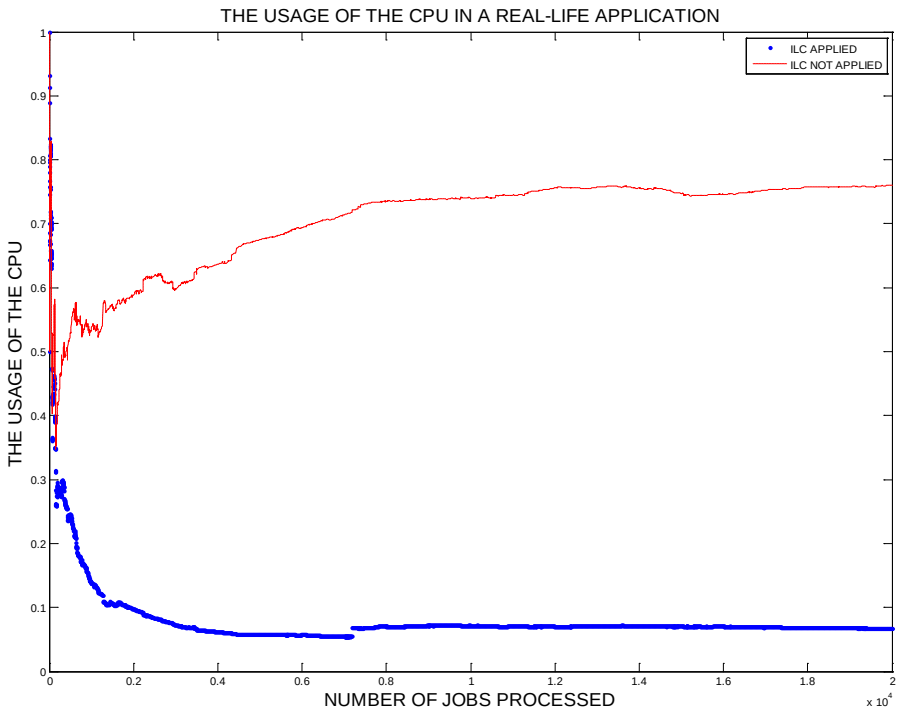


Figure 5.14 The Usage of CPU in a real-life application (Tenth Day)

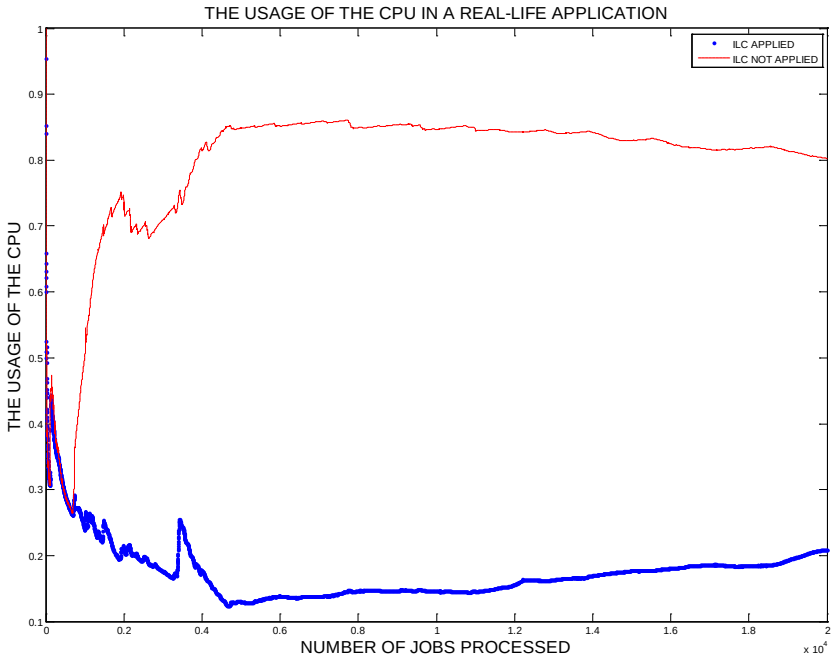


Figure 5.15 The Usage of CPU in a real-life application (Fifteenth Day)

In section 4, it is observed that when the jobs arrive to the computing system randomly the usage of the CPU varies between 0% and 25%. Through the arrival order of the jobs, in the real-life application of an individual retirement and life insurance company the usage of the CPU varies between 60% and 70%. Keeping outputs of the jobs in the memory provides gain about 20%-25%, but applying an ILC algorithm increases the gain to 75%-95%.

6. Future Works

In this paper, we tried to discover the gains of dynamic memory scheduling and we used single CPU system. It indicates that using double CPU-structures with one is split for I/O processes and the other one is split for intensive processes by Round-Robin scheduling algorithm that can improve the system performance [5].

Same jobs run periodically on some system. On these types of systems, periods can be observed and a part of the memory could be used for remembering jobs and periods. By this way, the requirements could be predicted and the system could get close the ideal form.

7. Conclusion

The experimental results demonstrated that an efficient dynamic memory allocation lead to remarkable improve on the system performance. In this study, we scheduled the idle memory to keep the outputs of repetitive jobs. To provide benefit from the methodology, a dynamic system developed.

In computer systems, the jobs run by the CPU with using the memory and the other system resources. In the enterprise applications, the outputs of the repetitive jobs are kept in the memory by the system developer. Thus the outputs of these jobs are brought from the memory without occupying the CPU. It is not analyzed on the jobs that kept in the memory how necessary they are. And also it pays no attention if there are necessary jobs that are not kept in the memory. The success of analyzes are controversial.

In our study the system defines its own requirements and keeps the outputs of the suitable jobs in the memory. In this way by the variable requirements on runtime, the system tries to get the maximum gain from the memory. It is observed that the study met with success.

Through our study, the developers will not have to decide which jobs should be kept in the memory. The improvements on the system performance and the response time were achieved. These improvements provide more rapid systems on the side of the end users.

8. References

[1] Ding, W., Guo, R., “Design and Evaluation of Sectional Real-Time Scheduling Algorithms Based on System Load”, *Young Computer Scientists*, 9, 14 - 18, (2008).

[2] Liu, C.L., Layland, J.W., “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”, *Journal of Association of Computing Machinery*, 20 (1), 46 - 61, (1973).

[3] Jensen, E.D., Locke, C.D., Takuda, H., “A Time-Driven Scheduling Model for Real-Time Operating Systems”, *Computer Science Department, Carnegie-Mellon University, Pittsburgh,* (1985).

[4] Chen, H., Xia, J., “A Real-Time Task Scheduling Algorithm Based on Dynamic Priority”, *Embedded Software and Systems*, 431 - 436, (2009).

[5] Kantabutra, S., Kornpitak, P., Naramittakapong, C., “Dynamic Clustering-Based Round-Robin Scheduling Algorithm”, *International Symposium on Communications and Information Technology*, 3, (2003).

[6] Nock, C., *Data Access Patterns: Database Interactions in Object-Oriented Applications*, Addison Wesley, Boston, (2003).

[7] Ford, C., Gileadi, I., Purba, S., Moerman, M., *Patterns for Performance and Operability – Building and Testing Enterprise Software*, Auerbach Publications, (2008).

[8] Butcher, M., Karimi, A., Longchamp, R., “A Statistical Analysis of Certain Iterative Learning Control Algorithms”, *International Journal of Control*, 81, 156 - 166, (2008).

[9] Al-Towaim, T., Barton, A.D., Lewin, P.L, Rogers, E., Owens, D.H., "Iterative Learning Control – 2D control systems from theory to application", *International Journal of Control (Special Issue: Multidimensional Control Systems: Theory with a view to Applications)*, 77 (9), 877 - 893, (2004).

[10] Moon, J., Doh, T., Chung, M.J., "An Iterative Learning Control Scheme for Manipulators", *Intelligent Robots and Systems*, 2, 759 - 765, (1997).

[11] Yi, W., Zhongs heng, H., Xingyi, L., "A Novel Automatic Train Operation Algorithm Based on Iterative Learning Control Theory", *Service Operations and Logistics, and Informatics*, 2, 1766 - 1770, (2008).

[12] Mi, C., Lin, H., Zhang, Y., "Iterative Learning Control of Antilock Breaking of Electric and Hybrid Vehicles", *Vehicular Technology*, 54 (2), 486 - 494, (2005).

[13] Xu, J., Wang, D., Wang, X., "The Analysis of Convergence Speed for an Open and Closed Loop Second Order Iterative Learning Control Algorithm", *Intelligent Control and Automotion*, 1, 3905 - 3909, (2006).

[14] Xu, J.X., Yan, R., "On Initial Conditions in Iterative Learning Control", *Automatic Control*, 50 (9), 1349 - 1354, (2005).

[15] Booth, P., Chadburn, R., Haberman, S., James, D., Khorasanee, Z., Plumb R.H., Rickayzen, B., *Modern Actuarial Theory and Practice*, Chapman & Hall/CRC, Florida, (2005).

[16] AvivaSA Time-Machine Application, http://crm.avivasa.com.tr/IceCreamWeb/, (2009).

[17] Bertsekas, D., Gallager, R., *Data Networks*, Prentice Hall, New Jersey, (1992).

[18] Pitman, J. *Probability*, Springer, Pittsburg, (1999).

[19] Ahn, H., Moore, K. L., Chen, Y., *Iterative Learning Control: Robustness and Monotonic Convergence for Interval Systems*, Springer, (2007).

[20] Machta, N., Bennani, M.T., Ahmed S.B., “Aspect Oriented Design of Real-Time Applications”, *Industrial Informatics*, 7, 763 - 767, (2009).

Biographical Sketch

Mutlu ERCAN was born in İstanbul in September 19, 1979. He graduated from İstanbul Orhan Cemal Fersoy Foreign Language Intensive High School in 1997. He received his B.S. degree in Computer Engineering in 2003 from Galatasaray University, İstanbul, Turkey.