INT J COMPUT COMMUN, ISSN 1841-9836
8(4):514-524, August, 2013.

Processing Capacity and Response Time Enhancement by Using Iterative Learning Approach with an Application to Insurance Policy Server Operation

M. Ercan, T. Acarman

Mutlu Ercan
Technology Management Department
AvivaSA Retirement and Life Insurance Company, İstanbul, Turkey
mutlu_ercan@yahoo.com

Tankut Acarman*
Computer Engineering Department
Galatasaray University, İstanbul,Turkey
tacarman@gsu.edu.tr
*Corresponding author

Abstract: In this study, computing system performance enhancement by using iterative learning technique is presented. Computational response time and throughput of the computing system is improved by introducing computational cost model and selection probability for each individual job. Excepted gain by enforcing dynamic caching is maximized in terms of classifying the arriving computing jobs on an selective manner and dynamically replacing them in a limited memory space. Gain maximization is performed by tuning the window size which helps to compare the computing jobs in terms of their individual selection and occurence probabilities. Fairly special computing work in insurance risk investigation is chosen for experimental validation of the proposed approach. Aspect Oriented Programming (AOP) methodology on Java platform is used for the experimental setup. AOP allows to identify computational jobs and their parameters based on the marked annotations. Experimental results show that the developed iterative learning based caching algorithm performs better than the other well known caching techniques.
Keywords: iterative learning, caching, computational cost model.

1 Introduction

Information processing requests and distributed applications are increasing along the web-based server-client interaction development cycle, and 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 is focussed on software control and dynamic resource allocation of the computing and storage units. Static and dynamic scheduling algorithms have been developed on a real-time basis systems operations. Static scheduling algorithms have low costs, but incapable to adapt scheduling rules to enhance efficiency of the limited resources subject to unpredicted computing jobs' arrival whereas dynamic scheduling algorithms may adapt and respond in a timely-fashion. The dynamic scheduling algorithms are mainly based on selection criteria driven approach such as Earliest Deadline First, Highest Value First, Highest Value Density First, [1]. Data access operations may degrade computing system performance. Unnecessary data reading may be avoided by caching the same and frequently occurring computational jobs, [2]. Caching is one of the common performance improvement patterns which may be applied in different layers of the systems, for example web browser caches static web content, database caches blocks of

data from disk for performance, the SAN controller will cache read/write data from disk and only complete when efficient to do so, (see for instance different caching approaches in [3]). There are some initial studies to model and control resources allocations in multi-tier virtualized applications which are considered to be part of the green and cloud computing. The applications are running in isolation by runtime resource allocation. In virtual execution environments, CPU utilization is controlled versus workload changes to achieve performance criterion based on mean response time, [4], [5]. A preliminary version of this paper is presented in [6] where the cost of processing a job in the Central Processor Unit (CPU) is derived and dynamic caching methodology is studied via simulation. Iterative Learning Control (ILC) 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, [7]. In factories, industrial manipulators performs same cycles repetitively, by using ILC energy consumption and operation time is minimized, [8].

In this paper, ILC-based caching scheme is introduced to enhance response time and processing capacity of the computing systems. Computational jobs are processed by the CPU which uses other storage devices such as memory and disk etc,. The service time of the CPU is minimized by preventing reprocessing the same jobs and results by enforcing learning scheme, time cost of obtaining the outputs of the jobs is minimized. Due to the limited caching memory resources, caching size is limited and a reference value for the iterative caching procedure is derived. The proposed dynamic caching scheme is experimentally tested on the insurance policy server operating on a real-time basis and its computational performance is presented. This paper contributes to dynamic caching for computing performance enhancement by using iterative learning technique.

This paper is organized as follows. In Section 2, computer process modeling and cost diagram of allocating the memory for caching and computational purposes is presented. The expected gain of the individual computational job is described. In Section 3, the reference value generation, the output value of the computing system and the error term is proposed towards autonomous management. Experimental studies to validate the effectiveness of the proposed approach are presented in Section 4. Finally, some conclusions are given.

2 Computing Process Modeling

From the view point of our study, the cost of processing a job in the CPU instead of caching its previously obtained result is considerably higher. The memory is used to store the repeated job results. Accessing to the memory from the CPU is fast enough. For example, the average time of getting an output from memory is approximately 12.10^{-7} milliseconds. A priori, cost of a simple request from the memory is cheaper than complex memory usages. CPU is the bottleneck where all jobs are processed, and mainly the memory is used to perform the jobs. There are two kinds of jobs: I/O jobs and computational jobs. The computational jobs are deterministic: they always give the same output with respect to the same input. "$3+x$" is a computational job, because when a value is set for "x" the output is independent from time. The other jobs are the I/O jobs for example writing to a file system. An I/O job such as data writing to a disk is not a deterministic job from this point of view. Random or time dependent jobs are considered to be I/O jobs. In this study, to minimize the time cost of processing the repeating jobs by a single CPU, some amount of the previously processed job results are cached in the memory. Obtaining the results from memory is a faster operation compared with running a complex job in CPU which may use other storage devices such as memory and disk etc,. Some partition of the memory is used to cache the job results and this partition versus the total memory space is

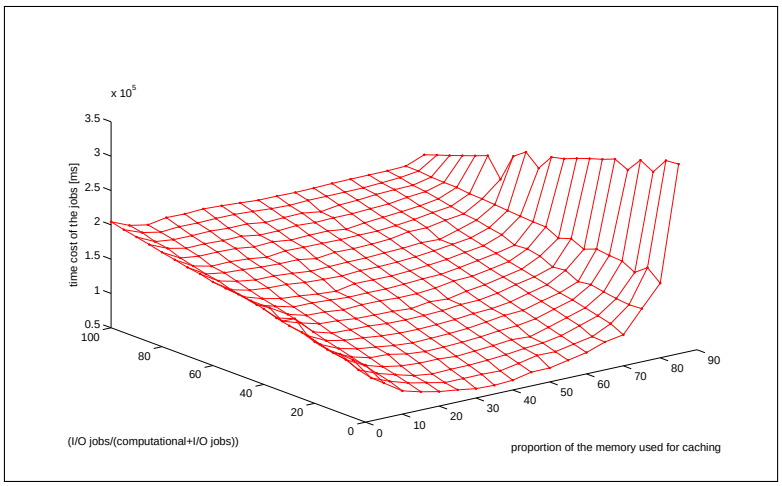


Figure 1: Cost diagram of memory allocation subject to static size to cache the jobs' results

denoted by α,

$$\alpha = \frac{m}{m_T} \tag{1}$$

where m denotes the total footprint of the job outputs and m_T is the total memory of the system. Each job produces an output object that occupies some footprint in the memory. The tradeoff of regulating the partition of the memory to cache some amount of the job results versus the request of memory space to process the job whose output has not been cached in the memory is illustrated by the following time cost diagram in Fig.1. In this figure, a set of I/O and computational jobs has been generated to investigate the computing performance of the system. The jobs are mainly constituted by I/O and computational tasks. The characteristics of the individual jobs such as footprints of the outputs, service and arrival rates are chosen randomly. The service rates are chosen between 1 and 10000, the footprints are between 1 and 1000 and all jobs are distributed uniformly. Time cost with respect to the memory caching partition and the ratio of the I/O jobs versus the total number of the I/O and computational jobs are plotted. When the small partition of the memory is reserved for caching purposes, the possibility of finding the previously processed result is low and the job needs to be reprocessed by the CPU which costs some time. If most of the jobs are I/O jobs, caching the computational job results does not affect the CPU processing time. When the number of computational jobs is increased versus the overall jobs, caching may reduce CPU's processing load by using the previously processed identical job result. On the other hand, to increase the memory caching partition may cause slower computing system response due to low memory space left for processing the new jobs or whose results have not been cached, *i.e.,* when the system becomes unable to find enough memory to process the new jobs, time cost of the jobs increases. The slope of the cost function presents a valley form and its slope in the left side illustrates this effect.

3 Real-time Dynamic Caching

The time cost of computation for each individual job may be described by,

$$t_{CPU}(\alpha) = \begin{cases} t_r & \text{if the job is operated by the CPU} \\ t_c & \text{if the job result exists in the memory} \end{cases} \tag{2}$$

where t_r and t_c denotes the time cost of the job run by the CPU (runtime) and the time cost of getting its output from the memory (cache time), respectively. The jobs are assumed to be in the

processing queue and the total size of the jobs is restricted by the window dimension. Initially, the last $n+m$ jobs can be supposed in the sliding window. When one job is processed, a new job enters in the sliding window and the last job is left out of the window as illustrated in Fig.2.

The total time cost of the computational jobs in the *kth* window is shown as;

$$\Delta t^k_{CPU}(\alpha) = \sum_{i=1}^{m} b_{ki} t^{ki}_{CPU}(\alpha) \qquad \forall b_{ki} \in \{0,1\} \tag{3}$$

b_{ki} denotes the existence of the *ith* job in the *kth* window. The total time cost of the I/O jobs in the same window,

$$\Delta t^k_{I/O}(\alpha) = \sum_{j=1}^{n} a_{kj} t^{kj}_{I/O}(\alpha) \qquad \forall a_{kj} \in \{0,1\} \tag{4}$$

a_{kj} denotes the existence of the *jth* job, m defines the number of computational jobs, n defines the number of I/O jobs in the *kth* window. Δt^k_{CPU} and $\Delta t^k_{I/O}$ are dependent on the reserved memory partition, denoted by α, to cache the previous job results as illustrated in Figure 2.

$$T^k(\alpha) = \Delta t^k_{CPU}(\alpha) + \Delta t^k_{I/O}(\alpha) \tag{5}$$

T^k is the total time cost of the *kth* window.

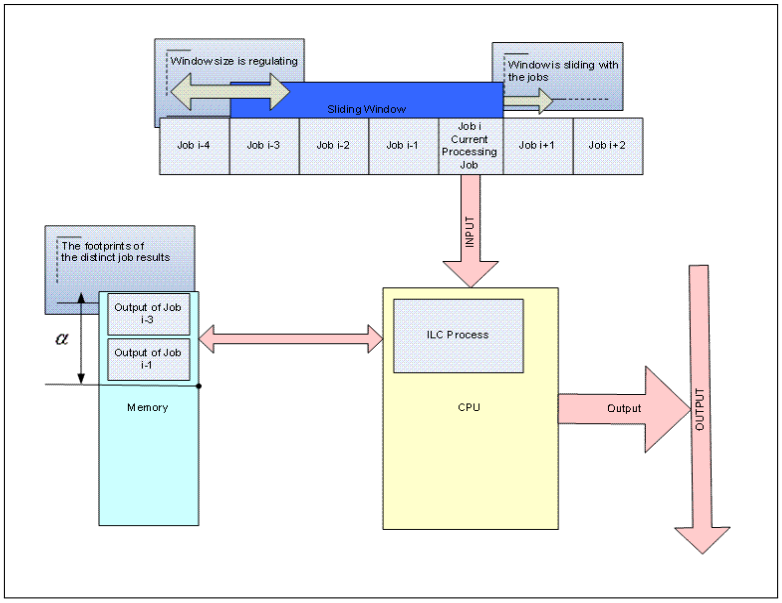


Figure 2: ILC-based caching scheme.

The proposed ILC-based dynamic caching scheme is illustrated in the Fig.2. ILC runs on the CPU as a memory allocator program and it selects some processed job results in the sliding window and caches them in the given partition of the memory. The sliding window size is adjusted in an optimal way to enable a performance criterion between forgetting the jobs with lower criteria and accepting the new jobs with higher selective criteria based on their recall or arrival frequency. In Fig. 3, selection scenario is illustrated where the window size is equal to 4, but the memory contains only 2 jobs which are selected based on the individual job's parameter. The results of the *(i-1)th* and *(i-3)th* computing job are selected and their results are cached in the memory to be used in their next recall. The selection parameter of the *(i-2)th* job is assumed to be lower compared to the cached ones. The selection parameter of the *(i+1)th* job is going to

be calculated in the next sampling period while the selection window is sliding towards the new coming jobs.

Probability of caching the individual job result depends on both of the probability on the selection criterion and the individual job's occurrence in the selection window which are denoted by $p_i^{sc}(\tau)$ and $\widetilde{p}_i(\tau)$, respectively.

$$p_i(\tau) = p_i^{sc}(\tau)\,\widetilde{p}_i(\tau) \tag{6}$$

The selection probability, denoted by p_i^{sc}, is calculated for each individual job and it is ranked among the other jobs to decide about caching the most suitable job results in the memory. The selection probability is derived by,

$$p_i^{sc}(\tau) = e^{-(\frac{am_i\mu_i\Gamma\tau}{\lambda_i C})} \tag{7}$$

where $\Gamma = \sum_{i=1}^{\Omega} \frac{1}{\mu_i}\tau$ is the selection (sliding) window size, m_i is the size of the individual job output, μ_i is the individual job service rate provided by the CPU, Ω is the number of distinct jobs in the computing system, λ_i is the arrival rate of the *ith* job, C is the capacity of the memory and a is the scaling factor to assure that the selective criterion takes values between 0 and 1. The job parameters are assumed to be determined *a priori*.

A simple analysis may show that when the individual size of the job output, denoted by m_i, increases, the selection probability p_i^{sc} decreases. This leads to cache as many as job results instead of caching few larger results. When the individual service rate, μ_i, increases, p_i^{sc} decreases leading to the choice of caching larger service time requiring jobs results instead of caching the jobs that can be processed faster. The selection criterion may increase as the individual arrival rate λ_i increases leading to cache the jobs with higher occurrence probability to avoid frequent computation efforts. When the sliding window size, τ is increased, the caching selection criterion p_i^{sc} for the individual job may be decreased due to the augmented number of the distinct jobs in the selection window.

The ratio of the *ith* individual computational job's arrival rate versus the sum of the arrival rate of the other individual jobs existing in the sliding window is given,

$$N_i = \frac{\lambda_i}{\sum\limits_{j=1}^{\Omega} \lambda_j} \tag{8}$$

For the individual job in the window, the probability of being selected by the ILC-based caching scheme is normalized with respect to the total number of the individual computational job's arrival rate ratio, $p_i = \frac{N_i}{N_{Total}}$ where $\mathrm{N}_{Total} \equiv \sum\limits_{j=1}^{N} \mathrm{N_j}$ and N is the total number of the processed jobs *i.e.,* the sum of the distinct jobs and the reprocessed ones. When the specified individual job is not in the sliding window, it may not be selected by the proposed ILC scheme. Through straight-forward analysis, the probability of not being selected due to absence of the individual job in the window,

$$\overset{*}{p}_i = 1 - p_i = 1 - \frac{\mathrm{N_i}}{\mathrm{N}_{Total}} = \frac{\mathrm{N}_{Total} - \mathrm{N_i}}{\mathrm{N}_{Total}} \tag{9}$$

Since the number of the jobs whose results may be nominated for caching is equal to the size of the window denoted by τ, the probability of being absent in the window is given by,

$$p_i(\tau) = \prod_{k=1}^{\tau} \frac{\mathrm{N}_{Total} - \mathrm{N_i} - k + 1}{\mathrm{N}_{Total} - k + 1} \tag{10}$$

Gain of the job is defined as the time cost of CPU to process its 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 of the individual job versus the sum of the absolute gains of the total jobs. The relative gain of the *ith* job in the window is defined by Ψ_i,

$$\Psi_i = \frac{\frac{1}{\mu_i}}{\Gamma} = \frac{1}{\mu_i \Gamma} \tag{11}$$

Total expected gain is obtained in terms of the average of all possible relative individual job gain weighted by its probability on existence in the sliding window and selection criterion, ([9]),

$$E(\Psi) = \sum_{i \ in \ \tau} \Psi_i \widetilde{p_i}(\tau) p_i^{sc}(\tau) \tag{12}$$

By using the derived probabilities given by and (7), (10) and (11), total expected gain for dynamic caching procedure may be given by,

$$E(\Psi) = \sum_{i=1}^{\Omega} \left(\frac{1}{\mu_i \Gamma} \left(\left(1 - \prod_{k=1}^{\tau} \frac{\mathrm{N}_{Total} - \mathrm{N}(i) - k + 1}{\mathrm{N}_{Total} - k + 1} \right) e^{-(\frac{a m_i \mu_i \Gamma \tau}{\lambda_i C})} \right) \right) \tag{13}$$

where Ω is the number of distinct jobs in the computing system. Total expected gain (13) is constituted by summing all the individual expected gain of the distinct jobs in the computing system. Towards ideal caching procedure, only the distinct job results are counted for the total expected gain, recalled and reprocessed jobs are not considered for the derivation of the expected gain.

The window size, denoted by τ, is regulated to maximize the expected gain of the sliding window by comparing the incoming jobs and caching the most profitable results for their possible future recalls which may lead to reduce computational efforts. The error variable is derived in terms of the attainable desired value, which is unknown *a priori* due to unknown job arrival, and the sum of the individual gains in the sliding window,

$$e_k = \frac{\max\limits_{\tau \in N}(E(\Psi)) - \sum_{i=k-\tau+1}^{k} \frac{\Psi_i p_i}{\tau}}{\max\limits_{\tau \in N}(E(\Psi))} = 1 - \frac{\sum_{i=k-\tau+1}^{k} \frac{\Psi_i p_i}{\tau}}{\max\limits_{\tau \in N}(E(\Psi))} \tag{14}$$

where N denotes the set of the previously arrived computing jobs, including the recalled and reprocessed ones, whose parameters such as occurrence frequencies, size, arrival rates may be learnt. The reference value is obtained by maximizing the derived total expected gain versus the sliding window size. The sliding window size is adjusted at each iteration step based on the previously processed and new arrival jobs to maximize the reference value for caching. The second term in the error variable is the sum of the individual job gains existing in the current sliding window standing for the actual output of the proposed caching system.

ILC methodology is based on the previously processed results and the job parameters such as their size in the memory, arrival and service rate, etc., and they will be used for the next memory scheduling process. The goal is to minimize the error and to regulate the window size to ensure caching the most suitable job results in the memory to avoid unnecessary computing efforts. To achieve this goal, the window dimension may be adjusted at each iteration step. The proposed ILC-based caching methodology is, (see for instance iterative learning regulator in [10] for different application)

$$\tau_{k+1} = \tau_k - sign(\frac{e_k - e_{k-1}}{\tau_k - \tau_{k-1}}) \tag{15}$$

Following (15) the window size is regulated at the next iteration based on the difference of two consecutive error terms and the window sizes denoted by e_k and τ_k at the kth iteration step, e_{k-1} and τ_{k-1} at the $(k-1)th$ iteration step, respectively. Ideally, the error variable in (14) may tend to zero if the given memory caching space is enough to cache all distinct job results.

4 Experimental Study

The proposed iterative learning based dynamic caching scheme is applied to the insurance policy server. This insurance company server executes more than 10.000 remote policy inquiries on a daily basis. To benefit from the individual retirement product, a customer must be more than 18 years old and 56 years old, and must be contributing to the insurance financial system more than 10 years. Most of the remote policy inquiring customers are between 18 and 46. About 350 customers are at the same age. At each age interval, half of them are male and female and they have to select their incomes and professions etc., in a limited set of the given query data. Many calculations are executed for each request coming from different individual customer having the similar input data.

In insurance and retirement enterprises area, the calculations of retention and premium need heavy actuarial computations. 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 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 is updated which may occur on a yearly basis, [11]. It can be assumed that for a life insurance policy with same input values always same premium is calculated. Aspect Oriented Programming

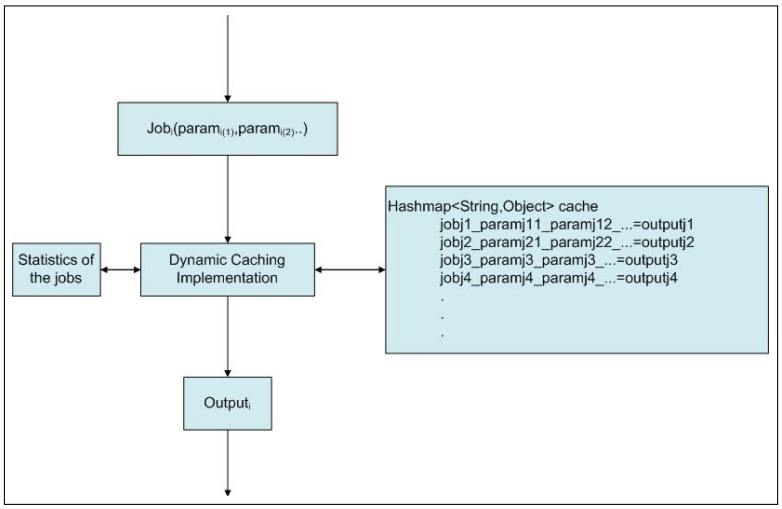
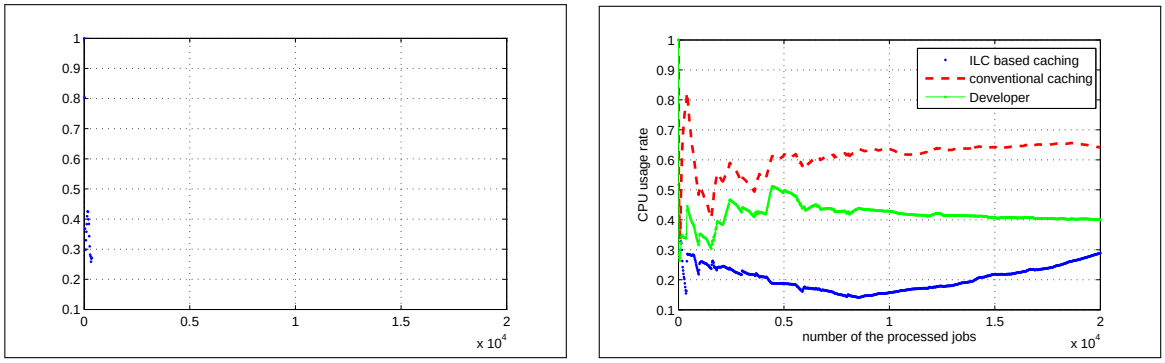


Figure 3: Implementation of process flow

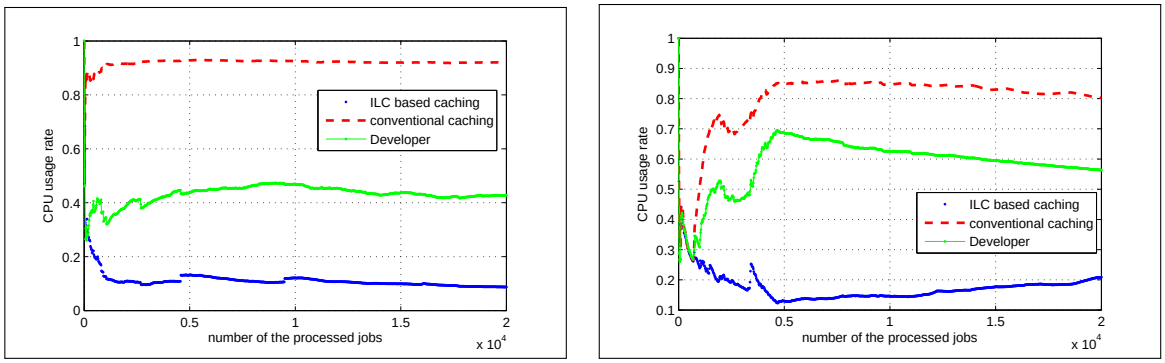
(AOP) methodology is used to separate business logic and other programming requirements such as logging, transactions etc. AOP works on join points like method calls, object instantiations with advice types; before, after, around etc. And also AOP is used to identify the individual jobs and their similar results. The developer marks computational jobs by annotations or regular expressions as join points. Each join point has its own signature, when a method is called by same parameters the signature does not change. If the output of the previous job is cached, it is accessible by the signature of the job, (see for instance [12] or more information on AOP

methodology). AOP has been designed to be implemented in many ways, including source-weaving or bytecode-weaving and directly in the virtual machine. Each method has its own signature including package and class name. In this study, a map is used to cache methods, key objects of the map are string implementation of the signature of the method concatenated with string implementation of the parameters (supposed that if and only if the parameters that have the same string implementation produce the same output). Implementation process is illustrated in Fig.3. The individual *ith* job is arrived with its parameters. The process searches whether the cache map has an object with string representation of the job (string representation consists of the signature of the method and the parameters separated by underscores). If the result has been obtained and cached for this string representation, the output is read from the cache and the parameters of the jobs are updated. Otherwise the job is processed by the CPU and the parameters are updated accordingly. In a real-time experiment, all of the jobs characteristics



(a) (b)

Figure 4: a) Process time of CPU to execute the incoming jobs on a real-time basis: The first day performance comparison of the CPU usage rate. b)The fifth day performance comparison.



(a) (b)

Figure 5: a) The tenth day performance comparison. b) The fifteenth day performance comparison.

are not known in advance. To overcome this situation, the system is designed as the average of the data characteristics cumulated in the previous days. The steady-state probabilities of the previous day distinct job parameters are used by the ILC-based caching scheme to calculate the overall job probabilities of the following day. In Fig.4 through Fig.5, CPU usage rate versus the large number of the jobs processed is plotted for different days. The performance of the proposed dynamic caching scheme is compared with other caching schemes. CPU usage rate responses are slightly changed dependent on the random arrival order of the remote policy inquiries. For

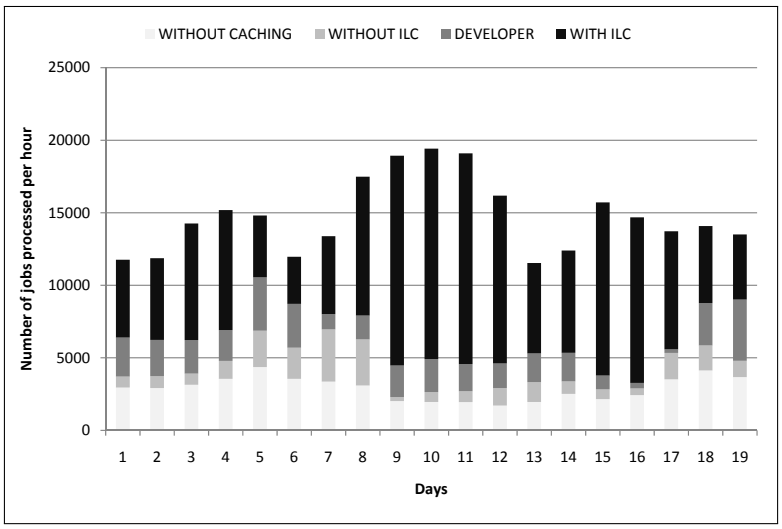


Figure 6: Comparison of the number of the jobs processed per hour for different caching schemes

each day, when CPU is enforced to process each individual job without considering even if their results are the same, CPU is loaded by %80 of its nominal capacity. When ILC is applied, the jobs are processed by using approximately 20% of the CPU computing capacity leading almost to %80 of free runtime. When the CPU has some free runtime, more jobs may be processed by

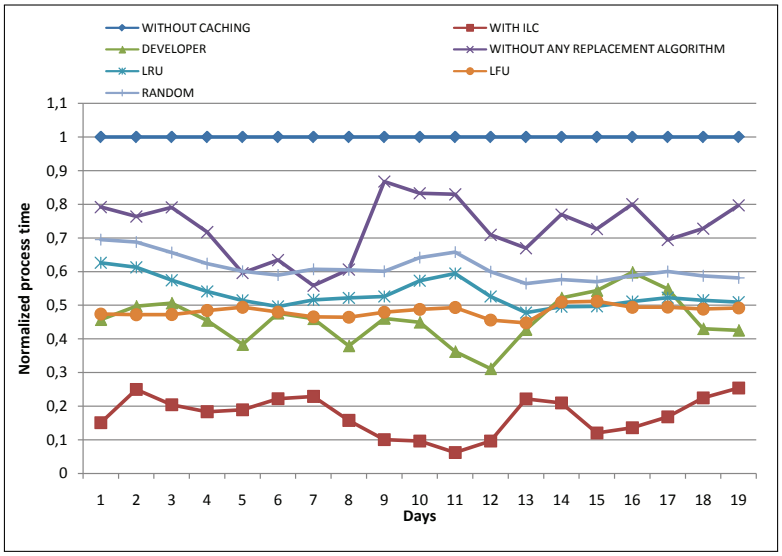


Figure 7: Comparison of the response time of the caching schemes

the proposed scheme.The number of the jobs that may be processed per hour by using the full CPU capacity is compared in Fig.6. For each day, ILC-based caching assures more jobs being processed per hour by utilizing the CPU's full capacity. Dark gray bars represent the number of the processed jobs by ILC-based caching. The lightest gray bar shows the number of the jobs processed without caching. When the first-in first-out caching scheme is applied, the number of the jobs processed is slightly increased, plotted with lighter gray bar. Even when the developer knows all parameters of the jobs, the job processing capability of the system per hour is less than the ILC-based system. The number of the processed jobs is significantly higher when the proposed learning and sliding window based caching algorithm is applied.

Different cache replacement algorithms such as random, least frequently used (LFU) or least recently used (LRU) are performed to compare with the introduced real-time ILC-based caching methodology. Different caching schemes react to the arrival job in a different manner such as when the computing job arrives, random algorithm chooses a random job result and replaces it in the cache, LRU chooses the least frequently arriving job or LRU chooses the job which has not been recalled for a long time. In Fig.7, processing time of the CPU to handle the incoming jobs per day is plotted for different caching systems. Without caching, the process time is considered to be the nominal and the process time of the other methodologies are normalized according to this nominal time.

The introduced ILC-based scheme, which avoids the unnecessary CPU computational efforts and permits the CPU react to the computational requests faster, is plotted with red-square line. The ILC-based system response time is compared with the system developer who is assumed to know all job characteristics, marks them and selects some of the results towards caching. The developer methodology time-response is around 40%, which is plotted with green-triangle line, and it is two times slower compared with ILC-based scheme. When random function is used for dynamic cache replacement method, process time is around 60% and it is plotted with solid and plus marked line. LRU, plotted with solid and x-marked line and LFU time responses, plotted with circle marked line, lead to respond two times slower compared to ILC-based caching, slightly slower than the developer's performance. The first-in first out caching methodology, which means without any replacement algorithm, causes fluctuations in the CPU response time and performs almost 20% faster at average compared to the case without caching.

5 Conclusion

In this paper, ILC-based caching scheme is introduced to enhance response time and processing capacity of the computing systems. The proposed scheme introduces the reference value for caching regulation. This reference value is introduced by summing possibly higher individual expected gains among the previously processed job results for the given window size. The average sum of the individual gain existing in the current window constitutes the actual output of the proposed caching system. Along the random arrival of the computational jobs, the size of the window helps to compare some amount of the computational jobs based on their parameters enabling a performance criterion about replacing the jobs with lower criteria by the jobs with higher selective criteria in the dedicated caching memory space. At each iteration step, the sliding window size is regulated to cache the most suitable job results in the memory based on the individual job selection probabilities. The sliding window size is incrementally increased and kept on inquiring the new arriving computational jobs until reaching to the operating point where the expected gain is maximized and the window size is adequately large to capture the most suitable job results subject to the given limited memory caching space.

To illustrate the effectiveness of the ILC-based caching system, the proposed dynamic caching scheme is applied to the insurance policy server. The proposed dynamic caching scheme is capable of processing more computing jobs for the given computing system capacity while the server being more responsiveness. By using the proposed autonomous caching and regulation methodology, the developers will not have to decide about the job results to be cached in the memory. Computing system performance, processing capacity and responsiveness to the new and unpredicted computational jobs will be autonomously improved without any expert intervention.

Bibliography

[1] Liu,C.L.; Layland, J.W. (1973); Scheduling Algorithms For Multiprogramming In A Hard-Real-Time Environment, *Journal of Association of Computing Machinery*, ISSN 1556-4665, 20(1): 46 - 61.

[2] Nock, C. (2003); *Data Access Patterns: Database Interactions In Object-Oriented Applications*,Addison Wesley.

[3] Ford, C. et al (2008) *Patterns For Performance And Operability – Building And Testing Enterprise Software*, Auerbach Publications.

[4] Padala, P. et al (2007), Adaptive Control of Virtualized Resources In Utility Computing Environments, *Proc. of the European Conference on Computer Systems*, Portugal:289-302.

[5] Kalyvianaki,E. et al (2009), Self-Adaptive And Self-Configured CPU Resource Provisioning For Virtualized Servers Using Kalman Filters, *Proc. of the 6th International Conference on Autonomic Computing*, New York, USA: 117-126.

[6] Ercan, M.; Acarman, T. (2010) Iterative Learning Control of Dynamic Memory Caching To Enhance Processing Performance On Java Platform, *International Conference on Computational Science*, Amsterdam, The Netherlands: 2664-2669.

[7] Butcher,M. et al (2008), A Statistical Analysis of Certain Iterative Learning Aontrol Algorithms,*International Journal of Control*, 81:156-166.

[8] Moon, J. et al (1997) An Iterative Learning Control Scheme For Manipulators, *Intelligent Robots and Systems*, 2:759-765.

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

[10] Ahn, H. et al (2007), Iterative Learning Control: *Robustness And Monotonic Convergence For Interval Systems*, Springer.

[11] Booth, P. et al (2005), *Modern Actuarial Theory And Practice*, Chapman & Hall/CRC.

[12] Clarke, S.; Baniassad, E. (2005) *Aspect-Oriented Analysis And Design: The Theme Approach*, New Jersey: Pearson Education.