Send Orders for Reprints to reprints@benthamscience.ae

An Automatic Adjustment Approach of Thread Quantity to Optimize Resource Usage

Zou Lida[1], Li Qingzhong[2,*] and Ma Yan[3]

[1,2]*School of Computer Science and Technology, Shandong University, Jinan, 250101, China*

[3]*State Grid Shandong Electric Power Research Institute, Jinan, 250002, China*

Abstract: In order to achieve optimization of resource usage for servers, a method is designed to automatically adjustment approach of thread quantity when multiple types of tasks are running, which can effectively and automatically optimize resource utilization without manual intervention. It first monitors different kinds of resources and trains resource usage of servers when adding a thread for a type of task. It then dynamically adds or reduces thread quantity to adapt to the scenario of dynamic change in resource usage according to monitoring results. When resources are idle and thread quantity needs to be added, the problem of determining thread quantity is abstracted by multi-dimensional container loading problem and we propose a heuristic algorithm based on similar ratio which can quickly obtain thread quantity. The proposed algorithm not only avoids the work of setting thread quantity for parallel tasks, but also improves the utilization rate of CPU, I/O and throughput.

Keywords: Automatic adjustment, resource utilization, server, thread quantity, throughput.

1. INTRODUCTION

For the server on which multiple tasks are concurrently running, the proposed method can reduce costs of acquisition and maintenance to fully use various resources. The determination of thread quantity is critical for resource utilization rate. The low thread quantity results in underutilization of resources, while high thread quantity causes too many resources consumption on unnecessary operations, such as thread switch and resource contention among threads. In order to optimize resource utilization for shared servers, it is necessary to determine and adjust thread quantity for each type of task. The adjustment of thread quantity is dynamic and complicated, which largely increases the workload of software developers. Thus, an automatic adjustment approach of thread quantity to optimize resource utilization is an urgent issue to be solved.

At present, there is some work on improving system performance by adjusting thread quantity. To perfect communication between server nodes [1], the approximate optimization on thread quantity was researched based on a round_robin scheduler in an operation system. In order to make full use of parallel computation in multi-core and multi-thread environment [2], an efficient algorithm which used bare threads with a set of optimal threads for allocation is exploited, which improves the speed-up ratio, computing and the real-time quality of the system. At present cloud computing platform and virtualization technology are widely used to change the mode of resource integration and resource usage. Virtualization technology [3] is suitable for the case when applications do not have enough loads to fill the peak of resources. In the paper we focus on adjusting thread quantity to achieve the dynamic optimization of resource usage in the scenario when application loads more or less match the configuration of servers. We also aim to reduce the workloads of software developers and improve developing efficiency.

2. ADJUSTMENT FRAMEWORK

In the section the framework of adjusting thread quantity to optimize resource usage is put forward. As shown in Fig. (**1**), adjustment framework consists of resource monitoring module, training module, determination module of thread quantity and thread management module.

Resource monitoring module monitors and records the usage of various resources. Training module first computes resource volume when a new thread is added for a type of

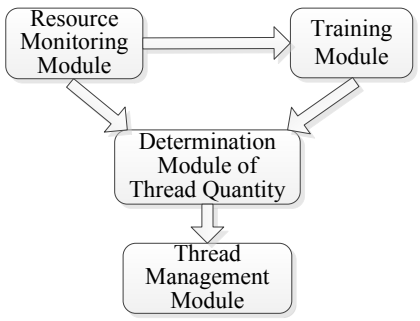

Fig. (1). Adjustment framework.

1874-4443/14

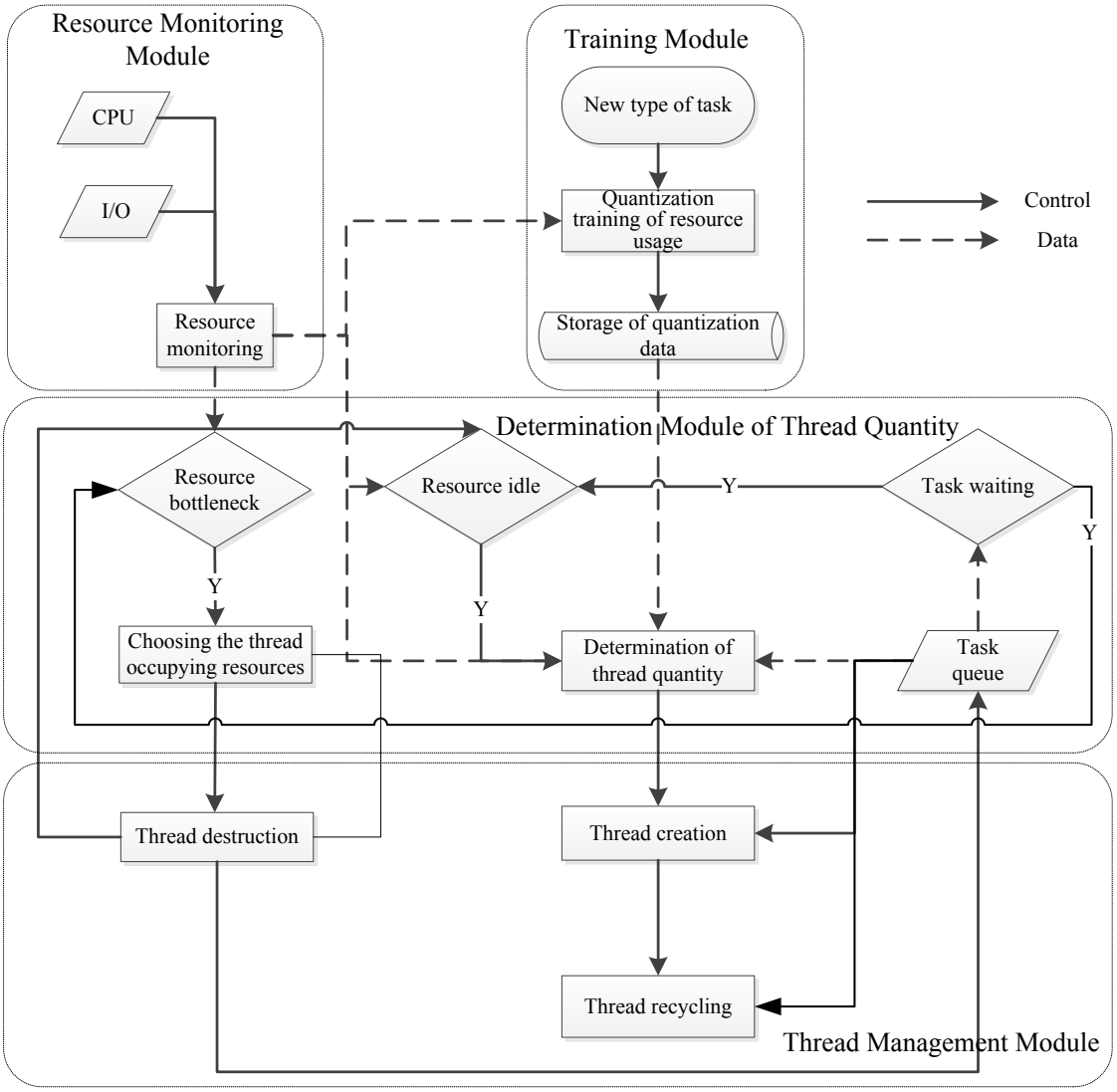

Fig. (2). The detailed flow diagram.

task according to monitoring results, and then stores the training data. Determination module of thread quantity obtains thread quantity for each type of task according to monitoring results and training data, and sends the adjusting instruction to thread management module. If resources are underutilized and there are waiting tasks in the queue, thread quantity is added. If resources are occupied by a task for a long time, thread quantity is reduced to release resources. Thread management module is in charge of thread creation, recycling and destruction according to the instruction from determination module.

3. AUTOMATIC ADJUSTMENT ALGORITHM OF THREAD QUANTITY

Based on the adjustment framework, our automatic adjustment algorithm is composed of four parts: 1) resource monitoring. 2) resource usage training. 3) dynamic adjustment of thread quantity. 4) thread management. The detailed flow diagram of algorithm is depicted in Fig. (**2**).

3.1. Resource Monitoring

The resource monitoring procedure in resource monitoring module monitors the usage of various resources. Taking CPU, I/O for example, the monitoring method is as follows. 1) The maximum capacity of each resource is tested and set to 1. 2) The resource usage is monitored in real time and then changed to percentage compared with the maximum capacity.

The monitoring information is respectively provided to the procedure of quantization training of resource usage and determination of thread quantity, resource bottleneck judger, resource idle judger.

3.2. Resource Usage Training

Quantization training of resource usage in training module quantifies resource usage volume of a thread for each type of task. Its quantization method is as follows: 1) It first initializes a process instance and creates several threads to

make it runnable for each type of parallel task. 2) It adds one thread at a time, and records resources increment. The computation method of resource increment is to compute the average resource increment in the whole lifetime of thread. If the task is long, we compute the average resource increment in a given time *t*. 3) It repeats the step 2) and adds *n* threads in all. 4) It computes the average value of resource increments on all the *n* threads for each resource. 5) It stores these average values. Its storage form is $R_i=\left[r_i^1 ... r_i^j ... r_i^m \right]$ for a type of task $T_i, i=1...l$, where r_i^j is the volume of resource *j* that a thread of task *i* occupies.

The storage data is provided to determine procedure of thread quantity

3.3. Dynamic Adjustment of Thread Quantity

Task queue is to create queue for each type of task. The newly arrived task is added to the right queue. Task queue provides the quantity of waiting tasks for task waiting judger.

Task waiting judger determines whether there are tasks waiting for execution and collects data in real time from task queue. If there are waiting tasks, resource bottleneck and resource idle judger are started; else the monitoring on task queue continues.

Resource bottleneck judger extracts monitoring data periodically and determines whether resources are in bottleneck state. The resource bottleneck state is the state when one or more resources reach their saturation levels and the other resources are underutilized. In fact, the ideal condition of resource usage is the case when all the resources are used evenly and fully. If resources are in bottleneck state, it triggers the procedure of choosing the thread occupying resources; else it continues to periodically estimate the resource state.

The procedure of choosing the thread occupying resources is to choose the thread that takes up excessive resource for a long time and causes resource bottleneck. It also sends instruction for thread destruction procedure.

Resource idle judger periodically extract monitoring data and determines whether resources are in idle state. If there are idle resources, it notifies determination procedure of thread quantity to compute and adjust thread quantity.

Determination procedure of thread quantity is responsible for obtaining and adjusting thread quantity. Its adjustment approach is as follows.

1. Resource idle threshold is set to L_j, that is, when utilization of resource *j* is lower than L_j, the resource is idle.
2. It computes the redundant volume of resources as $c_j = L_j - u_j$, where u_j is the current utilization.
3. Given $c_j, j=1...m$, $R_i=\left[r_i^1 ... r_i^j ... r_i^m \right], i=1...l$ and task queue, determination problem of thread quantity is converted to multi-dimensional container loading problem.
4. Multi-dimensional container loading problem is NP-hard. We next propose a greedy heuristic algorithm to solve it.
5. According to values obtained by heuristics, it sends the instruction of adding thread to the procedure of thread creation.

Problem Formulation. Dyckhoof *et al*. [4] summarize different kinds of container loading and cutting problem, which divide the problem into single-container and multiple-container problem, homogeneous, weak heterogeneous, strong heterogeneous problem. The boxes whose dimensions have the same direction and size are called the same type. The container loading problem with only one type of box is homogeneous. Weak heterogeneous container loading problem has several types of boxes but there are many boxes for each type. Strong heterogeneous container loading problem has many types of boxes but there are only several boxes for each type.

Kantorovich *et al*. [5] first present one-dimensional container loading problem, then many scholars [6, 7] propose approximation algorithms, such as FFD, BFD and NF. Paull *et al*. [8] proposes newspaper layout problem and two-dimensional container loading problem begins to be addressed [9, 10], such as bottom-left, bottom-left-fill, best-fit. With the development of logistics transportation, three-dimensional container loading problem causes concern and the integration or modification on simulated annealing, ant colony and genetic algorithm are presented to solve it [11, 12]. Our work belongs to single container, weak heterogeneous container loading problem. How many dimensions depend on the kinds of resources users pay attention to? If users focus on CPU and I/O, the determination of thread quantity is converted to two-dimensional container loading problem.

Based on the above analysis, we first formulize the problem. Given a container *C* and a set of types of boxes $B=\{b_1...b_l\}$, there are *n* boxes for each type and *n* is any large integer. The size of dimension in *C* is $c_1, c_2, ..., c_m$, which represents the redundant volume of different resources. The size of dimension in b_i is $r_i^1, r_i^2, ..., r_i^m$, which represents the usage volume of one thread for *m* resources. Assuming $x_i, i=1...l$ is the number of boxes for each type of box, the objective of the problem is to maximize fill rate of container. The fill rate is defined as $\lambda = S/V$, where the bulk of container is $V = c_1 c_2 ... c_m$ and the bulk of filled boxes is $S = \sum_{i=1}^{l} x_i r_i^1 r_i^2 ... r_i^m$.

Thus, the formulation description is $\max \lambda$

$$\text{s.t} \sum_{i=1}^{l} r_i^j x_i \le c_j, j=1...m \quad (1)$$

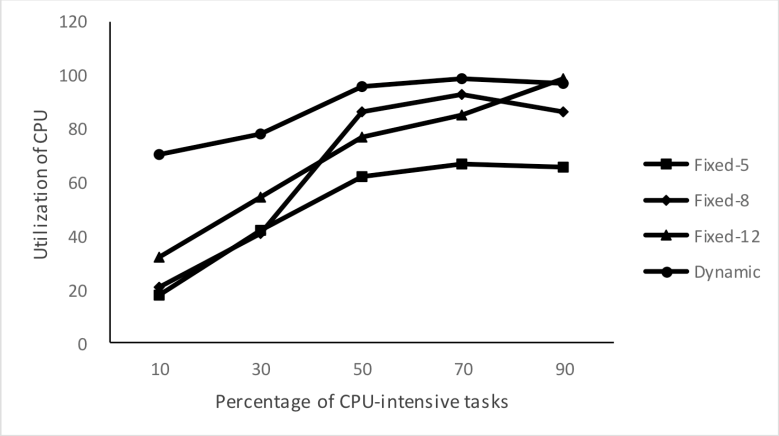

Fig. (3). CPU utilization rate with the ratio of CPU-intensive tasks.

Since container loading problem is NP-hard, heuristic algorithm is the first choice to solve it.

Algorithm 1 Heuristic Algorithm

Procedure H(*C,B*)
// *C* is container, *B* is the set of types of boxes.
1: while ($c_j \geq 0, j = 1...m$)
2: for *i=1* to *l*
3: compute
$d_i = \sqrt{(c_1 / c_j - r_i^1 / r_i^j)^2 + (c_2 / c_j - r_i^2 / r_i^j)^2 + ... + (c_m / c_j - r_i^m / r_i^j)^2}$
4: endfor
5: choose the type of box (denoted as $b_{i'}$) with the minimum d value
6: for *j=1* to *m*
7: $c_j = c_j - r_{i'}^j$
8: endfor
9: endwhile

Heuristic Algorithm. Our determination problem of thread quantity is different from regular container loading problem. In our problem how many dimensions depend on the number of resource types and it has not the constraints of stability. Hence we propose a novel greedy heuristic algorithm based on similar ratio. Its idea is as follows. Each time it selects a box with the most similar ratio with the redundant volume of container, until achieving the upper bound of each dimension in container.

First it normalizes the size of dimension for container and *l* types of boxes. For a given resource *j*, the other dimensions (i.e., the other resources) do the normalization based on this dimension of resource. Thus the size of dimension in *C* is $c_1 / c_j, ..., c_j / c_j, ..., c_m / c_j$, and the size of dimension in $b_i, i = 1...l$ is $r_i^1 / r_i^j, ..., r_i^j / r_i^j, ..., r_i^m / r_i^j$. Second it computes Euclidean distance between container and each type of box, i.e., $d_i = \sqrt{(c_1 / c_j - r_i^1 / r_i^j)^2 + ... + (c_m / c_j - r_i^m / r_i^j)^2}$. Each time it selects the type of box with the minimum d and puts an instance of this type in container. Accordingly, the size of dimension in *C* updates. Algorithm 1 gives the details of heuristic algorithm. Since the number of boxes for each type is any large integer, the set of *l* types of boxes remains unchanged.

3.4. Thread Management

Thread creation procedure is to create thread and gain task from task queue after receiving the instruction of adding thread. Thread destruction procedure is to destroy thread and put unfinished tasks back to task queue after receiving the instruction of destroying thread. It also notifies resource idle judger and determines whether there are idle resources after destroyed thread releases resource. Thread recycling procedure obtains new task from task queue to execute. If there is no waiting task, the thread is added to thread library to wait for task arrival.

In all, the proposed algorithm adjusts thread quantity through resource monitoring and thread management, which adapts to dynamic scenario of resource utilization. When resource usage is uneven, the thread that occupies resources in bottleneck state is destroyed to improve utilization. When resources are idle, threads are added to fully utilize resources. We also abstract the determination problem of thread quantity as multi-dimensional container loading problem, which not only achieves automatic and quick adjustment of thread quantity without manual intervention, but also reduces the workload of software developers.

4. EXPERIMENTAL RESULTS

We next test the performance of proposed algorithm. We compare our algorithm with a fixed number of threads. The indicators of performance are chosen as CPU utilization, I/O utilization and throughput. According to the often used thread quantity, the fixed number of threads is randomly selected from 5 to 20. We select three times and thread quantity is respectively 5, 8, 12. The experimental server is IBM p690. Both computation-intensive tasks and I/O-intensive tasks are chosen. Each time 10 task instances are running and we gradually increase the ratio of computation-intensive tasks. The fixed number of threads is denoted as *Fix-5, Fix-8, Fix-12* while our algorithm is called *dynamic*.

We first compare CPU utilization rate of different algorithms. As shown in Fig. (**3**), the proposed algorithm *dynamic* has higher CPU utilization rate, especially in high ratio of I/O-intensive tasks. Since threads are occupied by I/O-intensive tasks, *dynamic* algorithm can still add threads to

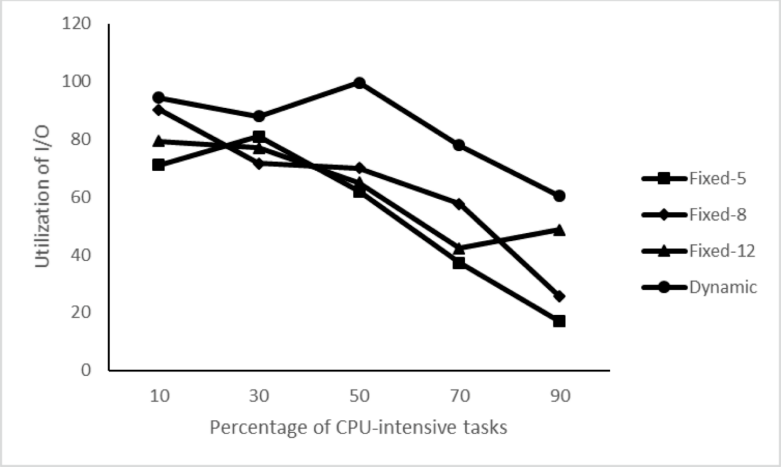

Fig. (4). I/O utilization rate with the ratio of CPU-intensive tasks.

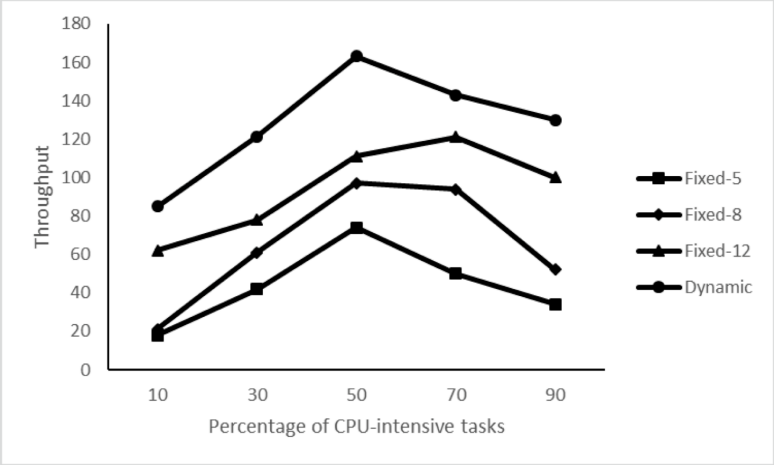

Fig. (5). System throughput with the ratio of CPU-intensive tasks.

execute computation-intensive tasks, while fixed algorithms can only wait for I/O tasks to be completed and this causes CPU to be idle. When CPU-intensive tasks account for 90%, *Fixed-12* has similar CPU utilization since *Fixed-12* could fully use CPU and does not cause block.

We then validate the utilization of I/O for different algorithms. From Fig. (**4**) we see that *dynamic* algorithm performs best especially in high ratio of computation-intensive tasks. Our proposed algorithm can flexibly add threads to execute I/O-intensive tasks according to the idle capacity of I/O.

We last compare system throughput of different algorithms. In Fig. (**5**) *dynamic* algorithm has higher throughput compared with the case of fixed thread quantity. Since *dynamic* algorithm could dynamically add thread quantity according to real-time usage of resources, the resources of servers can be fully utilized and the overall system throughput is improved.

CONCLUSION

In this paper we use resource monitoring and thread management to achieve automatic adjustment of thread quantity without manual intervention, which also reduces the workload of software developers. We abstract the determination problem of thread quantity as multi-dimensional container loading problem and propose a heuristic algorithm, which realizes dynamic, accurate and quick adjustment of thread quantity and is feasibly optimum or approximate optimum of resource utilization. The extensive experiments show that the proposed algorithm performs better than the case in fixed number of threads on CPU utilization, I/O utilization and throughput for different types of tasks.

CONFLICT OF INTEREST

The author confirms that this article content has no conflict of interest.

ACKNOWLEDGEMENTS

This work was supported by the National Natural Science Foundation of China under Grant No.61272241, No. 61303085, No. 61303005.

REFERENCES

[1] C. Zuo, X. Liu, X. Chen and D. Liu, “Approximate optimization thread quantity of inner communication in distributed parallel serv-

er", *Journal of Harbin Engineering University*, vol. 26, no.5, pp. 614-618, 2006, In Chinese.

[2] K. Wang and L. Wang, "Research on optimal thread quantity of real-time signals acquisition system", *Journal of Computer Applications*, vol. 31, pp. 2593-2596, 2011, In Chinese.

[3] X. Fei, L. Fangming, J. Hai, and A. V. Vasilakos, "Managing performance overhead of virtual machines in cloud computing: a survey, state of the art, and future directions", In: *Proceedings of the IEEE*, vol. 102, 2014, pp. 11-31.

[4] H. Dyckhoff and U. Finke, "Cutting and Packing in Production and Distribution", *A typology and bibliography*, Springer, 1992.

[5] L. V. Kantorovich, "Mathematical methods of organizing and planning production", *Management Science*, vol. 6, pp. 366-422, 1960.

[6] G. Belov and G. Scheithauer, "A cutting plane algorithm for the one-dimensional cutting stock problem with multiple stock lengths", *European Journal of Operational Research*, vol. 141, pp. 274-294, 2002.

[7] Xavier and F. K. Miyazawa, "A one-dimensional bin packing problem with shelf divisions", *Electronic Notes in Discrete Mathematics*, vol. 19, pp. 329-335, 2005.

[8] A. Paull, Linear programming, "A key to optimum newsprint production", *Pulp and Paper Magazine of Canada*, vol. 57, pp. 85-90, 1956.

[9] M. Hifi, T. Saadi, and N. Haddadou, "High performance peer-to-peer distributed computing with application to constrained two-dimensional guillotine cutting problem", in *Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, 2011, pp. 552-559.

[10] K. He, W. Huang and Y. Jin, "Efficent algorithm based on action space for solving the 2d rectangular packing problem", *Journal of Software*, vol. 23, no. 5, pp. 1037-1044, 2012, In Chinese.

[11] D. Zhang, Y. Peng and L. Zhang, "A multi-layer heuristic search algorithm for three dimensional container loading problem", *Chinese Journal of Computers*, vol. 35, no. 12, pp. 2553-2560, 2012, In Chinese

[12] C. D. Tarantilis, E. E. Zachariadis, and C. T. Kiranoudis, "A hybrid metaheuristic algorithm for the integrated vehicle routing and three-dimensional container-loading problem", *IEEE Transactions on Intelligent Transportation Systems*, vol. 10, 255-271, 2009.

Received: September 22, 2014 Revised: November 03, 2014 Accepted: November 06, 2014