Advances in Computer Science and Information Technology (ACSIT)
Print ISSN: 2393-9907; Online ISSN: 2393-9915; Volume 2, Number 4; April-June, 2015 pp. 355-361
© Krishi Sanskriti Publications
http://www.krishisanskriti.org/acsit.html

Agent Based Storage Compaction and BST Load Balancing Algorithm for Multicore Architecture

G. Muneeswari[1] and J. Frank Vijay[2]

[1]Department of Information Technology SSN College of Engineering
[2]Department of Information Technology KCG College of Technology
E-mail: [1]muneeswarig@ssn.edu.in, [2]hodit@kcgcollege.com

Abstract— *In a Multicore architecture, besides enormous performance enhancement, lot of challenges are injected on the operating system storage compaction and load balancing point of view. The main objective of agent based system is to invent some methodologies that make the developer to build complex systems that can be used to solve sophisticated problems. In this paper, we proposed the time based storage compaction algorithm for multicore architecture. Another research issue in multicore system is the development of effective techniques for distributing workload on multiple processors. To improve the load balancing a new BST based load balancing technique has been proposed. We actually simulated this algorithm in the linux kernel 2.6.11 and the results show that it improves the speedup and performance of the multicore processors to 20%.*

Keywords: *Multicore; garbage collectors; storage compaction; inductive learning; BST; Load balancing*

1. INTRODUCTION

Multicore architectures, which integrate several processors on a single chip, are being widely accepted as a solution to serial execution problems currently limiting single processor system designs. In most proposed multicore architectures (fig.1), different cores share the global common memory. High performance on multicore processor requires that storage compaction has to be effectively reinvented.

Traditional storage compaction algorithms focuses on collecting unwanted files (garbage) on a single processor or might be implemented for distributed systems. The similar kind of storage compaction algorithms can be extended for multicore systems thus increasing the space in the memory and ultimately improving the cpu performance. Multi-core processors do, however, present a new challenge that will need to be met if they are to live up to expectations. Since multiple cores are most efficiently used (and cost effective) when each is executing one process, simultaneously many processes can be kept in the memory for execution. As the number of cores per processor and the number of threaded applications increase, the performance of more and more applications will be limited by the processor's memory availability. Storage compaction in today's operating systems have the primary goal of collecting all the unwanted files, delete them and thus keeping all cores busy executing some runnable process. One technique that mitigates the memory limitation is to intelligently collect the garbage files and make the memory free with the help of software approach like agent based system, which incorporates inductive learning.

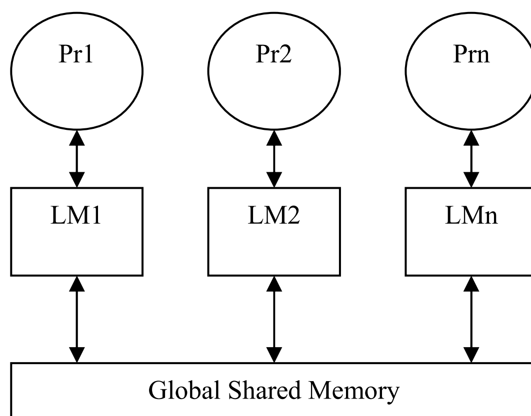

Fig. 1: Multicore Architecture

The main goal is to allocate the processes to processors to maximize throughput, maintain stability, resource utilization and should be fault tolerant in nature. Load balance is critical for performance in large multicore systems. If there is a load imbalance on multiple processors then it can cause hundreds and thousands of cores to be kept in idle state. Improving load balance requires a detailed understanding of the amount of the load per processor and an insight into the arrival rate of the tasks must be known to the scheduler. Most of the modern load balance mechanisms are often integrated into applications and make implicit assumptions about the load.

The three load balancing steps are:

- Evaluate the imbalance;
- Decide how to balance if needed;
- Reallocate work to correct the imbalance.

To address the first two requirements with a data structure called as PSIB and derive complete information with a help of Binary Search Tree (BST) and the load is balanced based on the tree construction.

The Paper is organized as follows. Section II reviews related work. In Section III, we introduce the garbage collector (storage compaction) concepts. This describes local garbage collector implementation and BST load balancing algorithm implementation. In section IV, We discuss the evaluation and results. The section V presents conclusion. Finally, section VI describes future enhancements with multicore.

2. BACKGROUND AND RELATED WORK

Bacon and Rajan [1] discovered similarities between forward-tracing and reference-counting uniprocessor collectors, noting that optimized versions of each collector behave similarly and have similar performance traits because they seem to be composed of the same underlying tracing actions. Beltway [2] composed a unique mechanism for dividing the heap by object age and performing incremental collection with high performance. In [3] uniprocessor collectors are extended to a distributed context.The train algorithm [4] has been instantiated with PMOS and distributed heaps, which is termed to be DPMOS mechanism. The Doomsday distributed termination [5] detection protocol also used for the efficient applications.

Lowry [6] discusses the safe and complete distributed garbage collection with the train algorithm. Moreau [7] presents a formal proof of reference listing by introducing a graphical representation of the algorithm's state space and permitted transitions therein. In [8], The pseudo root approach of the distributed garbage collection for mobile actor systems is explained. Munro [9] describes the selection policies using the PMOS garbage collector. Norcross [10] describes the construction of train based collectors by the composition of Distributed Termination Detection Algorithm (DTDA) and arbitrary local collector.

Zigman [11] discusses the creation of compound collectors by composing multiple collectors to operate on subgraphs. Past research into partition selection [12] has focused on heuristics aimed at reclaiming acyclic data structures by selecting partitions that contain objects that are the targets of erased pointers.Herlihy and Moss [13] presented the first algorithm for lock free garbage collection in a realistic model. The algorithm assumes that processes synchronize by applying read, write and compare & swap operations to shared memory.

3. STORAGE COMPACTION AND BST LOAD BALANCING

Garbage Collectors

The data items or files which are allocated or created, but not being used for long time, will stay in memory, wasting some useful space of the memory. These waste data items or files which are called garbage can be detected and can be freed from the memory space, can be used for some other useful data items or files. The entire idea forms the basis for the storage compaction.

Automatic storage management in high level languages saves the programmer from the time consuming and error prone task of manually managing the allocation and de-allocation of storage space. Instead, the language runtime systems abstract over the underlying storage mechanisms by dynamically allocating space and automatically reclaiming it when it is no longer used by the application. As with garbage identification, there are two techniques underlying any garbage reclamation scheme. Either each live object is copied to some part of the managed storage space, where it is guaranteed to be maintained or each garbage object is directly reclaimed and added to a free list. The way in which space is reclaimed is directly associated with the mechanisms by which space is allocated;

The system model (taken from [14]) is defined such that a computation executes over a number of sites where each site acts independently, concurrently and asynchronously. The following assertions are made:

1. Each site has its own local storage and communicates with other sites only through message passing.
2. Local storage is dynamically allocated and automatically (safely) reclaimed.
3. Sites appear to operate correctly, without Byzantine behavior.
4. There is no bound on the relative rates of computation of the sites.
5. Events at a given site are totally ordered; since messages are delivered only after being sent, events are partially ordered in the system as a whole.
6. Messages are delivered in-order, without omission or corruption.

This concept of collecting the garbage is the motivation for designing these local and global garbage collector algorithms. The local garbage collector at each node collects the files which are not used for some specific time period and the global collector checks all the local garbage files and finally collects the files which are garbage to the whole multicore environment.

Local Garbage Collector

The local garbage collector consists of two components. They are local garbage collection algorithm and a local garbage collection agent. The local garbage collector along with the processor and memory interface is shown in the fig.2. In a multicore environment, every processor is sharing the common global memory and the processors are allocated with a local garbage collector.

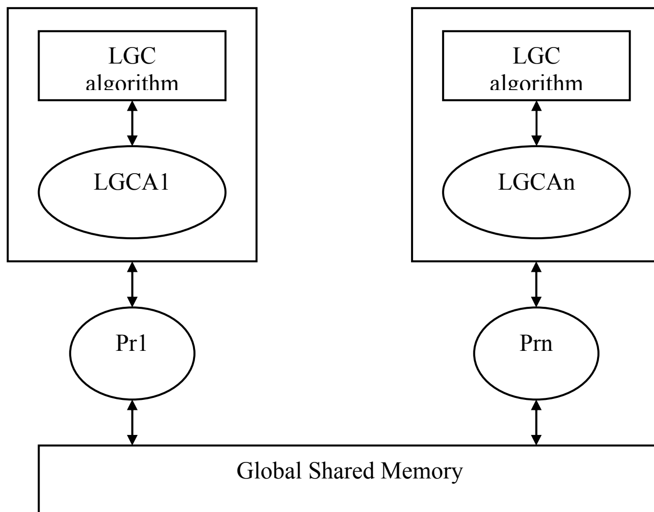

Fig. 2: Local garbage Collector model

The significant components of this module includes the time based algorithm and a software agent which is described in detail in the next sections.

Local Garbage Collection Algorithm

This algorithm is implemented in such a way that the user has to give the accessing time of the files in each core. The user can select any number of files that he wants to access. Initially the flags of all the files are set with initial time of access and then whenever the file is opened for access, the flag is reset with the current time. The timer routine implemented in the local garbage collection algorithm is depicted as follows:

```
Int timer()
{       int ct_time,t1m;
        struct time t;
        gettime(&t);
        t1m = t.ti_min;
        ct_time = t1m;
        return(ct_time);
}
```

When the algorithm reaches the target time, it will check which of the files are accessed between the initial and target time. These files are said to be the non garbage files and they need not be freed from the memory. The remaining files in the multicore system are said to be the garbage files, which are local to the individual processor.

Local Garbage Collection Agent

Once the local garbage collection is completed, the LGCA at each processor constructs a data structure that implements a linked list. The individual node in the linked list consists of the following components:

The name of the file that are not accessed at the corresponding processor.

The file access bit indicates the processor access. If the file access bit is 1, it emphasize that the processor has not accessed the file.

Otherwise the bit is set to 0.

Initially according to the specified algorithm, the local garbage collection agent constructs the linked list with the starting set of garbage files. As the global garbage collector finds the global set of files, the local agent deletes the global garbage files and modifies all the remaining files access bit as 0. Later, when again the LGCA is invoked then the local agent searches for the garbage files in the linked list. If it is available then it simply makes the file access bit as 1.Finally the entire list is searched for the file access bit whose entry is 0. Those identified nodes are no longer been left in the list and has to be removed.

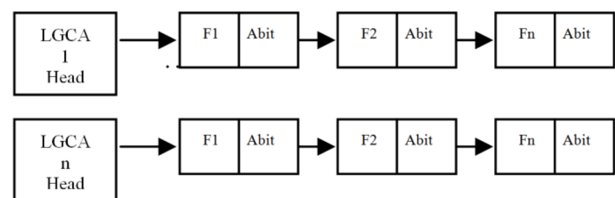

Fig. 3: Local Garbage Collection Agent Model

In the same way the global garbage collection also performed.

Load Balancing System Model and Algorithm Construction

If the online temporary task assignment problem is considered, every process has an arrival and a departure time. The main objective is to assign the jobs such that the maximum load over both machines and time is minimized. It is shown that no polynomial time algorithm can achieve an approximation ratio below 1:5 for this problem. However, for the case where the number of machines is getting increased, the load balancing becomes a complex issue. In online load balancing with unrelated machines that is specifically in heterogeneous multicore system it is very difficult to balance the load because every machine has different processing capability.

Proposed Policies

In multiprocessor system often the load balancing algorithm preempts jobs and migrate jobs between different processors. The following Table.1 illustrates the different policies incorporated in this approach.

Table 1: Policies adopted in agent based load balancing

decision making invocation	event driven (application arrival)
transfer policy	local information only (threshold)
location policy	least loaded
acceptance policy	single requestno rejection allowed
information policy	periodic state dissemination

BST Agent based Load balancing Algorithm

This algorithm based on the binary search tree construction for the multicore architecture.

Step 1: Processor State Information Block (PSIB) is constructed by the operating system.
Step 2: Threshold time period α is set by the scheduler for every processor.
Step 3: After α time period the PSIB has been verified by the load balancing agent.
Step 4: PSIB generally contains the processor ID and the load on every core.
Step 5: The load balancing agent obtains load of all the processors from the PSIB.
Step 6: Load balancing agent constructs the BST with two different data. (i.e.) processor ID and the load on every core.
Step 7: Now if a new process arrives into the system, load balancing agent refers the BST for the lightly loaded node.
Step 8: The left most node in the BST is the lightly loaded node and hence that will be chosen for new load allocation.

Algorithm Analysis

With each test that fails to find a match at the probed position, the search is continued with one or other of the two sub-intervals, each at most half the size. More precisely, if the number of items, N, is odd then both sub-intervals will contain (N - 1)/2 elements, while if N is even then the two sub-intervals contain N/2 - 1 and N/2 elements. If the original number of items is N then after the first iteration there will be at most N/2 items remaining, then at most N/4 items, at most N/8 items, and so on. In the worst case, when the value is not in the list, the algorithm must continue iterating until the span has been made empty; this will have taken at most $\lfloor \log_2(N) + 1 \rfloor$ iterations, where the $\lfloor\ \rfloor$ notation denotes the floor function that rounds its argument down to an integer. This worst case analysis is tight: for any N there exists a query that takes exactly $\lfloor \log_2(N) + 1 \rfloor$ iterations. When compared to linear search, whose worst-case behavior is N iterations, it is seen that binary search is substantially faster as N grows large. For example, to search a list of one million items takes as many as one million iterations with linear search, but never more than twenty iterations with binary search. However, a binary search can only be performed if the list is in sorted order.

Average performance

$\log_2(N)-1$ is the expected number of probes in an average successful search, and the worst case is $\log_2(N)$, just one more probe. If the list is empty, no probes at all are made. Thus binary search is a logarithmic algorithm and executes in O (log(N)) time. In most cases it is considerably faster than a linear search. It can be implemented using iteration, or recursion. In some languages it is more elegantly expressed recursively; however, in some C-based languages tail recursion is not eliminated and the recursive version requires more stack space. Binary search can interact poorly with the memory hierarchy (i.e. caching), because of its random-access nature. For in-memory searching, if the span to be searched is small, a linear search may have superior performance simply because it exhibits better locality of reference. For external searching, care must be taken or each of the first several probes will lead to a disk seek. A common method is to abandon binary searching for linear searching as soon as the size of the remaining span falls below a small value such as 8 or 16 or even more in recent computers. The exact value depends entirely on the machine running the algorithm.

It is seen that for multiple searches with a fixed value for N, then (with the appropriate regard for integer division), the first iteration always selects the middle element at N/2, and the second always selects either N/4 or 3N/4, and so on. Thus if the array's key values are in some sort of slow storage (on a disc file, in virtual memory, not in the cpu's on-chip memory), keeping those three keys in a local array for a special preliminary search will avoid accessing widely separated memory. Escalating to seven or fifteen such values will allow further levels at not much cost in storage. On the other hand, if the searches are frequent and not separated by much other activity, the computer's various storage control features will more or less automatically promote frequently accessed elements into faster storage. When multiple binary searches are to be performed for the same key in related lists, fractional cascading can be used to speed up successive searches after the first one. Even though in theory binary search is almost always faster than linear search, in practice even on medium sized arrays (around 100 items or less) it might be infeasible to ever use binary search. On larger arrays, it only makes sense to binary search if the number of searches is large enough, because the initial time to sort the array is comparable to many linear searches.

4. EVALUATION AND RESULTS

In this section, we present a performance analysis of our storage compaction algorithm using a gcc compiler and storage compaction model is taken from linux kernal version 2.6.11. The results show that there is a linear increase in the cpu performance as we increase the free space results in global garbage collection. Our algorithm results in allocating more processes (since we deleted unwanted files) and keeping the processor busy and reduces the average waiting time of the processes in the centralized queue. For our simulation we have taken 10 files as a sample (training set) and tested against 3 cores. In Fig.4, the cpu performance against the free space is shown for the proposed inductive learning based storage compaction algorithm. We discovered that the average performance of the processors in the multicore environment increases to 20% as we increase the free space in the memory.

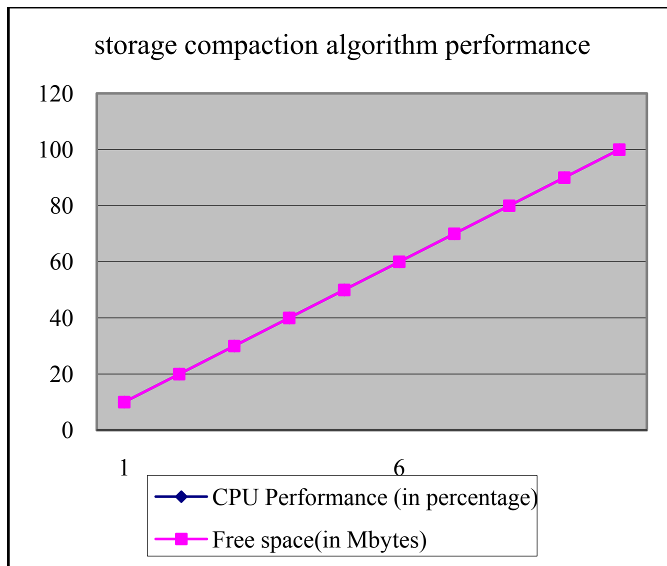

Fig. 4: Storage Compaction Algorithm Performance

The overall performance factors of load balancing policies like Round Robin, Random, Local Queue, Central Manager, Threshold and the BST agent approach are listed below in Table.2 Actually, the process were varied from 5 to 165 keeping the number of cores as constant 25. The performance factors prove that the load balancing algorithm improves the performance by 0.35% on an average compared to other algorithms given with standard workload benchmarks.

Table 2: Performance factors for different load balancing algorithms

Process	Round robin perf Factor	Random perf Factor	Local Queue perf Factor	BST Agent perf Factor
5	0.823521	0.509345	0.636455	0.874285
10	0.751399	0.465306	0.54516	0.864447
15	0.710574	0.527285	0.515193	0.825197
20	0.670876	0.487493	0.521377	0.86367
25	0.753355	0.549243	0.635469	0.771012
30	0.748557	0.423462	0.507126	0.82059
35	0.757628	0.496635	0.61542	0.862243
40	0.748389	0.459606	0.54692	0.875972
45	0.777301	0.501497	0.654516	0.825484
50	0.756046	0.462384	0.567489	0.842653
55	0.749089	0.511923	0.586223	0.873321
60	0.763368	0.489332	0.618097	0.843654
65	0.732016	0.574167	0.574165	0.865062
70	0.792815	0.390873	0.585458	0.870745

The Comparative analysis of Round Robin, Random, Local Queue, Central Manager, Threshold policies and the BST Agent approach are listed below in Table 6.4. From the analysis it is observed that the average waiting time and average turnaround time are much lesser (0.2%) compared to other load balancing policies. The performance of various load balancing algorithms is measured by the following parameters.

- Overload Rejection

If Load Balancing is not possible additional overload rejection measures are needed. When the overload situation ends then first the overload rejection measures are stopped. After a short guard period Load Balancing is also closed down.

- Fault Tolerant

This parameter gives that algorithm is able to tolerate tortuous faults or not. It enables an algorithm to continue operating properly in the event of some failure. If the performance of algorithm decreases, the decrease is proportional to the seriousness of the failure, even a small failure can cause total failure in load balancing.

- Forecasting Accuracy

Forecasting is the degree of conformity of calculated results to its actual value that will be generated after execution. The static algorithms provide more accuracy than of dynamic algorithms as in former most assumptions are made during compile time and in later this is done during execution.

- Stability

Stability can be characterized in terms of the delays in the transfer of information between processors and the gains in the load balancing algorithm by obtaining faster performance by a specified amount of time.

- Centralized or Decentralized

Centralized schemes store global information at a designated node. All sender or receiver nodes access the designated node to calculate the amount of load-transfers and also to check that tasks are to be sent to or received from. In a distributed load balancing, every node executes balancing separately. The idle nodes can obtain load during runtime from a shared global queue of processes.

- Nature of Load Balancing Algorithms

Static load balancing assigns load to nodes probabilistically or deterministically without consideration of runtime events. It is generally impossible to make predictions of arrival times of loads and processing times required for future loads. On the other hand, in a dynamic load balancing the load distribution is made during run-time based on current processing rates and network condition. A DLB policy can use either local or global information.

- Cooperative

This parameter gives that whether processors share information between them in making the process allocation decision other are not during execution. What this parameter defines is the extent of independence that each processor has in concluding that how should it can use its own resources.

Table 3: Parameters for different load balancing algorithms

Parameters	Round Robin	Random	Local Queue	BST Agent	Central Manager	Threshold
Context switch overhead	No	No	Yes	No	No	No
Scheduling Ratio	More	More	Less	More	More	More
Average Waiting Time	Less	More	Less	Less	More	Less
Average Turnaround Time	Less	More	Less	Less	More	Less
Accuracy	More	More	Less	More	More	More
Delay	Large	Large	Small	Small	Large	Large
Centralized/ Decentralized	D	D	D	C	C	D
Dynamic/Static	S	S	DY	DY	S	S
Cooperative	No	No	Yes	Yes	Yes	Yes
Process Migration	No	No	Yes	No	No	No
CPU Utilization	Less	Less	More	More	Less	Less

- Process Migration

Process migration parameter provides when does a system decide to export a process? It decides whether to create it locally or create it on a remote processing element. The algorithm is capable to decide that it should make changes of load distribution during execution of process or not.

- Resource Utilization

Resource utilization include automatic load balancing A distributed system may have unexpected number of processes that demand more processing power. If the algorithm is capable to utilize resources, they can be moved to under loaded processors more efficiently

5. CONCLUSION

Although the results from the linux kernal version 2.6.11 analysis in the previous section are encouraging, there are many open questions. Even though the improvement (cpu performance) possible with number of cores, for some workloads there is a limitation by the following properties of the hardware: the high off-chip memory bandwidth, the high cost to migrate a process, the small aggregate size of on-chip memory, and the limited ability of the software (agents) to control hardware caches for deleting a file. We expect future multicores to adjust some of these properties in favor of our time based storage compaction and BST load balancing algorithm. Future multicore will likely have a larger ratio of compute cycles to off-chip memory bandwidth and can produce better results with our algorithm.

6. FUTURE ENHANCEMENTS

This paper has argued that multicore processors pose unique free space management problems that require an agent based software approach that utilizes the large number processors very effectively. We also proved that lot of drastic enhancements in the traditional garbage collector part of operating system that optimizes for cpu cycle utilization. We discovered that the cpu performance increases slowly with the increase of free space. As a conclusion our new novel approach eliminates the complexity of collecting the garbage files in the many core systems and improved the cpu utilization to the maximum level since we employ a new novel agent based BST load balancing algorithm.

REFERENCES

[1] D.F.Bacon, P.Chang and V.T.rajan. A Unified theory of garbage collection SIGPLAN Notices, Volume 39, Number 10, pages 50-68, 2004.

[2] S.M.Blackburn, R.E. Jones, K.S.Mckinley and J.E.B.Moss Beltway. Getting around garbage collection gridlock. In Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation, Programming Language Design and Implementation (PLDI), Berlin, June 2002, Volume 37(5) of ACM SIGPLAN Notices, ACM Press 2002.

[3] S.M.Blackburn, R.L.Hudson, R.Morrison, J.E.B.Moss, D.S.Munro and J.Zigman, Starting with termination: A methodology for buildig distributed garbage collection algorithms. In Proceedings Australasian Computer Science Conference 2001.

[4] W.F.Brodie – Tyrrell, H. Detmold, K.E.Falkner and D.S.Munro. Grabage Collection for Storage-Oriented Clusters. In Conferences in Research and Practice in Information Technology, Volume 26, pages 99-108, Dunedin, Newzealand, 2004.

[5] M.Linesey, R.Morrison, and D.S.Munro. The Doomsday Distributed Termination Detection Protocol. In Distributed Computing, Volume 19, pages 419-431. Springer 2006.

[6] M.C.Lowry and D.S.Munro. Safe and Complete Distributed Garbage Collection with the Train Algorithm. In Proceedings of International Conference on parallel and Distributed Sytems. ICPADS' 02, pages 651-658, Taipei, Taiwan, Dec.2012.

[7] L.Moreau, P.Dickman and R.E.Jones. Birrell's distributed refernce listing revisited. ACM transaction on Programming Language System, Volume 27, Number 6, pages 1344-1395, 2005.

[8] W.Wang and C. A. Varela. Distributed garbage collection for mobile actor systems: The pseudo root approach. Technical Report 06-04, Dept. of Computer Science, R.P.I., Feb. 2006. Extended Version of the GPC'06 Paper.

[9] R.Morrison, D.Balasubramsniam, R.M.Greenwood, G.N.C.Kirby, K.Mayes, D.Munro and B.C.Warboys. A compliant persistent architecture. Software, Practice and Experience, Volume 30, Number4, pages 363-386, 2000.

[10] D.S.Munro and A.L.Brown. Evaluating partition selection policies using the CMOS garbage collector. In A.Dearle, G.Kirby and D. Sjoberg (editors), POSG ninth International workshop on Persistent Objects Systems, pages 104-115, Lille hammer, Norway, September 2000.

[11] S.Norcross. Deriving Distributed Garbage Collectors from Distributed Termination Algorithm.Ph.D thesis.

[12] J.N.Zigman, A General Framework for the description and construction of Hierarchical garbage collection algorithms. Ph.D thesis, Australian National University, June 2004.

[13] Maurice P.Herlihy, J.Eliot B.Moss, Lock-Free Garbage Collection for Multiprocessors, IEEE Transactions on Parallel and Distributed Systems, Vol.3, No.3, May 1992.

[14] S.J. Norcross, R. Morrison, D.S. Munro, and H. Detmold, "Implementing a Family of Distributed Garbage Collectors", in *26th Australasian Computer Science Conference(ACSC 2003)*, Adelaide, Australia*, p. 161-170* (2003)

[15] Ali M. Alakeel, "Load Balancing in Distributed Computer Systems", International Journal of Computer Science and Information Security Vol. 8 No. 4 July, 2010.

[16] Dahoud Ali, Mohamed A. Belal and Moh'd Belal Zoubi, "Load Balancing of Distributed Systems Based on Multiple Ant Colonies Optimization" , American Journal of Applied Sciences 7 (3): 433-438,2010.

[17] G.Muneeeswari, K.L.Shunmuganathan, A Novel hard-soft processor affinity scheduling for multicore architecture using multiagents, European journal of Scientific Research,Vol.55, No.3, PP 419-429, 2011.

[18] G.Muneeeswari, K.L.Shunmuganathan, Agent Based Load Balancing Scheme using Affinity Processor Scheduling for Multicore Architectures,WSEAS Transactions on Computers,August 2011.

[19] G.Muneeswari, A.Sobitha Ahila, Dr.K.L.Shunmuganathan, "A Novel Approach to Multiagent Based Scheduling for Multicore Architecture", GSTF journal on computing, Singapore vol1.No.2, 2011.

[20] G.Muneeswari, Dr.K.L.Shunmuganathan, "Improving CPU Performance and Equalizing Power Consumption for Multicore Processors in Agent Based Process Scheduling", International conference on power electronics and instrumentation engineering, Springer-LNCS, 2011.

[21] Agus Dwi Suarjaya , "A New Algorithm for Data Compression Optimization", International Journal of Advanced Computer Science and Applications, Vol. 3, No.8, 2012.

[22] G.Muneeswari, K.L.Shunmuganathan, "Time based agent garbage collection algorithm for multicore architectures", ICACCI '12 Proceedings of the International Conference on Advances in Computing, Communications and Informatics,2012.

[23]