International Journal of Engineering Trends and Technology
ISSN: 2231 – 5381 /doi:10.14445/22315381/IJETT-V70I3P207

Volume 70 Issue 3, 54-65, March, 2022

Original Article

Conservative Dynamic Load Balancer for Performance Enhancement of NUMA Multiprocessor Systems

D. A. Mehta[1], Priyesh Kanungo[2]

[1]*Professor, Shri G S Institute of Technology and Science, Indore, MP, India*
[2]*Head, Computer Centre, Devi Ahilya Vishwavidyalaya, Indore, M.P, India.*

[1]mehta_da@hotmail.com

Abstract *- In pursuit of enhancing the performance of NUMA multiprocessor systems in terms of throughput, CPU utilization and Turn Around Time of processes, Linux load balancer performs load balancing periodically, which in turn causes storms of load balancing attempts and process migrations involving large overheads of time. Many of these attempts are futile and impose performance penalties. We, therefore, propose a Conservative Dynamic Load Balancer which avoids the aggressive load balancing as done by existing load balancers and adheres to the restrictive policies of balancing the load under certain conditions. Reducing the overheads of load balancing attempts, process migration, and memory & cache access improves the Turn Around Time of processes significantly as compared to the Linux load balancer. The results of experimentation exhibit the performance gain in the range of 7-12 % for different NUMA systems.*

Keywords *- Dynamic Load Balancing, DLB, Load Balancer, NUMA, Sched domain***.**

I. INTRODUCTION

Multiprocessor and Multicore systems are typically designed based on Non-Uniform Memory Access (NUMA) architecture. A NUMA Multiprocessor/ Multicore system (NUMA system) is organized in the form of Nodes. A node consisting of a set of processors (the terms processor and core are used interchangeably in this paper), part of the main memory and I/O, placed on a common bus, is connected to other nodes via some high speed, high bandwidth interconnection network. Memory in a particular node is at a *distance* (which refers to latency, bandwidth or hops) from the processors of other nodes, resulting in the non-uniform access time of local and remote memories [1] [20]. A typical NUMA system is shown in Figure 1. It is said to have 2 Memory Access Levels (MALs) due to two different memory latencies:

(i) When a processor accesses memory in its own node.
(ii) When the processor accesses any memory outside its node [5].

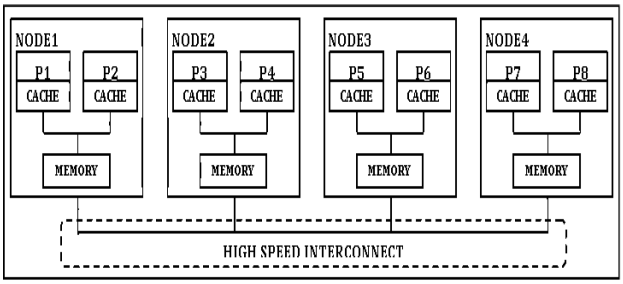

Fig. 1 NUMA system with 4 nodes, 8 Processors and two memory access levels
(P1, P2 … are Processor1, Processor2 …)

A. Dynamic Load Balancing

Linux, a widely used operating system for NUMA systems, implements separate run queues for each processor and, to avoid any load imbalance among them, incorporates a Dynamic Load Balancing (DLB) technique in the scheduler. Its load balancer makes use of a data structure 'sched domain', which groups processors together in a hierarchy that mimics the physical hardware. A scheduling domain or sched domain is a set of processors which share properties and scheduling policies. Figure 2 depicts the sched domain hierarchy for the system shown in Figure 1.

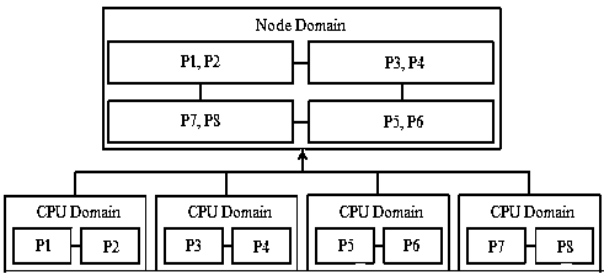

Fig. 2 Sched domain hierarchy for NUMA system with two memory access levels

The lowest level sched domains are called CPU/Core domains. Each CPU domain consists of all processors of a particular node and points to a higher domain (parent domain) called node domain which consists of this particular node and all those nodes which are at some

particular *distance* from this node [5] [9]. Thus, for a NUMA system with two memory access levels, there will be two levels in the sched domain hierarchy, and the node domain will comprise all the nodes of the system, as shown in Figure 2. The sched domain hierarchy defines the scope of load balancing for each processor. In a scheduling domain, the sets of processors among which the load balancing is performed are called scheduling groups. For a processor performing load balancing at the lowest level domain, all the processors in its node will be the scheduling groups; and at higher levels, all the nodes at that level will be the scheduling groups. Load balancer, which runs on each processor separately, is invoked in three different situations and performs the load balancing as explained below [15] [17] [23]:

- Periodically at specific time intervals: During the periodic load balancing cycle, the load balancer traverses the entire sched domain hierarchy, starting at the current processor's sched domain, and initiates a balancing operation if it is due for balancing. At each level, it first finds the busiest processor of the busiest scheduling group and then migrates the tasks (processes or threads) from that processor to the current processor if the load of the busiest processor is more than the load of the current processor, as per the load threshold (25%; or 12% in some cases).
- When a task is newly created or woke-up through system calls fork(), exec(), wakeup(): In this condition, the task is moved to the least loaded processor of the least loaded scheduling group (node) in its current domain.
- When a processor becomes idle: In this condition, *idle load balancing* is performed by the idle processor; it selects the most loaded scheduling group in its current domain and migrates tasks from the most loaded processor to this processor.

It is evident from the foregoing description that large overheads of time are involved in performing the dynamic load balancing. Though these overheads are inevitable and not avoidable always, an efficient load balancer should minimize them by finding the conditions under which unnecessary attempts of load balancing and process migrations may be avoided. The objective of our work, therefore, is to design such an efficient load balancer to improve the performance of NUMA systems.

II. AN ANALYSIS OF LOAD BALANCING ATTEMPTS AND PROCESS MIGRATIONS DONE BY EXISTING LINUX LOAD BALANCER

In every load balancing cycle, the Linux load balancer executing on each processor performs a *Load Balancing Attempt* wherein it tries to find a processor in the current scheduling domain which is more loaded than the current processor. If any such processor is found, processes are migrated from that processor to the current processor to balance the load, and this attempt is called a successful load balancing attempt; if no processor overloaded as compared to the current processor is found, the attempt is called an unsuccessful load balancing attempt [6] [16]. The load balancer carries out this process of load balancing for all the scheduling domains in the sched domain hierarchy. Apart from the periodic load balancing cycle, *idle load balancing* is also performed in the same manner when any processor becomes idle [10] [12].

While performing the load balancing in this manner, many loads balancing attempts and consequent process migrations succeed. However, a significantly large no. of attempts and/or process migrations fail too. Moreover, many successful attempts prove to be unfruitful also. All such unsuccessful or unfruitful load balancing operations result in situations that are undesirable from the performance point of view.

Reasons for such undesirable situations are described below, along with the experimental results (which are the outcome of the experimentation performed over a variety of NUMA multiprocessor systems with various types of workloads) to substantiate our reasoning/inferences.

A. Unsuccessful Load Balancing Attempt

Load Balancing (LB) attempt is made by the load balancer, but it remains unsuccessful since the busiest processor of the busiest scheduling group under consideration is not found overloaded as compared to the current processor. The possible reason could be that, after the last load balancing cycle (during which the system's load was balanced)-

- No process might have exited or entered the system, and hence a load of all processors is almost balanced.
- Even if a new process has entered the system, it is assigned the least loaded processor, and load balance is maintained.
- In case a load of any processor would have become zero, it must have initiated an *idle load balancing* operation and pulled the appropriate no. of processes from the overloaded processors to balance the load.

Due to aforesaid reasons, many attempts of load balancing done during the lifespan of the processes remain unsuccessful, as evident from Table 1, which shows few representative cases based on the experimental results. It is noticeable from this table that out of the total load balancing attempts done by the load balancer, a fairly large no. of attempts remain unsuccessful and thus result in huge unnecessary time overheads.

Table 1. Unsuccessful load balancing attempts in linux for the NUMA systems and the workloads mentioned

NUMA System Architecture: No. of Nodes-Processors per Node-No. of MALs	**Workload:** Type of processes; No. of processes arriving randomly; Av. Execu. time of each process	**LB Attempts**		
		Total No. of LB Att. done	*No. of LB Att. not succeeded*	*% of LB Att. not succeeded*
16-2-6	CPU bound; 50; 500 ms	244	211	86.48
16-2-6	CPU bound; 100; 500 ms	270	182	67.41
16-2-6	CPU bound; 150; 500 ms	345	263	76.23
16-2-6	CPU bound; 200; 500 ms	497	299	60.16
16-2-6	CPU bound; 400; 500 ms	804	422	52.49
16-2-6	CPU bound; 200; 200 ms	238	109	45.80
8-4-6	CPU bound; 100; 300 ms	213	152	71.36
8-2-3	CPUbound;100; 300 ms	148	88	59.46
8-2-3	CPU bound; 400; 300 ms	519	277	53.37
8-2-3	IO bound; 200; 300 ms	390	140	35.90

B. Successful Load Balancing Attempt but Unsuccessful Process Migration

Even if certain load balancing attempts are successful and consequently the process migrations take place, the migrated processes are not executed on the destination processors- they are further migrated from those processors to some other processors, may be large no. of times- proving the migrations to be unsuccessful effectively.

The possible reasons of unsuccessful process migrations could be:

a) The processor on which some processes have been migrated is already loaded heavily, and hence few recently migrated processes do not get scheduled for a long time, maybe till the next load balancing cycle, and get migrated from this processor to some other less loaded processor in that cycle.

b) An IO bound processes environment where the run queue of a processor may seem to be overloaded, and therefore processes are pulled from such processors; however, after a short time, IO-bound processes to proceed for IO operations, reduce the load of that processor and cause it to pull back the previously migrated processes.

c) Some processes have very large burst time and/or very low priority.

It is therefore apparent that even if the load balancing attempts are successful, the consequent process migrations prove to be unnecessary and add to the undesirable overheads.

Following Traces of a few representative processes out of 400 processes that were executed on some NUMA systems depict the unnecessary migrations.

Process-255: P10.wait(1384) - P10.run(79)- IO(21)- P10.wait(1680) - P10.run(77)- IO(23)-P10.wait(219)- **P11.wait(361)- P10.wait(34)- P9.wait(313)- P8.wait(113)-** P5.wait(25) - P5.run(100) -P5.wait(0)- P5.run(29).

Process-109: P1.wait(32)- **P6.wait(29)-** P12.wait(442) - P12.run(100) -P12.wait(2186) - P12.run(100) - P12.wait(2264) - P12.run(78)- IO(22)-P12.wait(756)- P12.run(37).

Process-374: P0.wait(165)- **P12.wait(1473)- P6.wait(20)- P0.wait(442)- P3.wait(362)- P2.wait(396)-** P0.wait(210) - P0.run(100) -P0.wait(407)- P3.wait(663) - P3.run(100) - P3.wait(432) - P3.run(100) -P3.wait(324)- P3.run(25).

Process-283:P10.wait(1126)- P10.run(71)- IO(29)- P10.wait(722)- **P11.wait(967)- P5.wait(195)- P1.wait(33)- P10.wait(33)- P11.wait(266)- P10.wait(41)- P11.wait(669)-** P4.wait(831) - P4.run(100) -P4.wait(196)- P3.wait(129) - P3.run(100) -P3.wait(104)- P3.run(60).

Process-167: P24.wait(75)- **P26.wait(205)- P43.wait(137)-** P37.wait(237) - P37.run(100) - P37.wait(247) - P37.run(100) -P37.wait(54)- **P61.wait(88)-** P62.wait(148) - P62.run(100) - P62.wait(21)- P62.run(61).

Process-106: P22.wait(331)- **P15.wait(202)- P22.wait(138)-** P50.wait(115) - P50.run(50)- IO(50)- P50.wait(100) - P50.run(77)- IO(23)-P50.wait(73)- P53.wait(403) - P53.run(62)- IO(38)-P53.wait(16)- P53.run(47).

The bold portions in the trace of each process show that this particular process was unnecessarily migrated across a few processors. For instance, Process-106 was originated on processor P22; it got migrated to processor P15; waited in its run queue for a certain amount of time and then got migrated back to processor P22; it did not get executed on P22 and, after some time, got migrated to processor P50. Trace of Process-374 shows that it was unnecessarily migrated 05 times from one processor to another and eventually returned back to the parent processor. Process-283 hops 09 times from one processor to another processor. Likewise, many other processes were also unnecessarily migrated.

It is obvious that in this kind of scenario, the load balancer incurs the overheads of load balancing attempts as well as that of process migrations, which prove to be unnecessary.

C. Successful Load Balancing Attempt & Successful Process Migration, but not Advantageous

In many cases, a process is migrated and also gets executed on the destination processor, but its memory pages may be lying on the node from which it is migrated or on a far node. This situation arises because the load balancer while migrating the processes, does not pay any attention to the origin of the process.

From the simulation results, it was noted that a significant no. of processes were migrated to and executed on the nodes far away from their parent nodes, as shown in Table 2.

Table 2. No. of processes executed on nodes belonging to particular level of sched domain hierarchy in NUMA multiprocessor system (having Linux Load Balancer)

Level of sched domain hierarchy	Processes executed on Nodes of this level	
	No. of such processes	*% of such processes*
I	87	43.5 %
II	27	13.5 %
III	23	11.5 %
IV	21	10.5 %
V	15	07.5 %
VI	27	13.5 %

It can be observed that out of 200 processes executed on a NUMA system having 16 nodes, 32 processors and 06 Memory Access Levels, 31.5 % of processes were executed for a long period of their life span, on far nodes- the nodes belonging to IV, V and VI levels of sched domain hierarchy; 22 % processes were executed, on very far nodes- the nodes belonging to V and VI levels of the hierarchy. For such processes, indirect overheads of process migration, i.e., the increased memory latencies and cache-miss overheads, outweigh the advantages of load balancing.

Experimentation was done, and the memory access time was analyzed for both the cases- when load balancing was done for all the levels and when it was restricted to a few levels, and it was found that even if the slight imbalance is caused in the latter case, the overall average TAT of processes is better due to reduction in memory access time and cache-miss overheads.

The foregoing analysis and discussion gives an insight into the functioning of the Linux load balancer and reveals an important point: ‘’it is possible to improve the load balancing performance by not performing load balancing in certain conditions’’. This key point became the basis for designing an efficient load balancing algorithm, as described in section IV.

III. RELATED WORK

The performance improvement of load balancing algorithms largely revolves around minimizing the unnecessary load balancing attempts and process migrations. Focusing on how to perform the load balancing judiciously, many researchers have suggested approaches to minimize the load balancing overheads and to improve the performance thereby.

Lim et al., for instance, suggested an approach to minimize the cost of task migration by considering the importance level of running tasks on multicore embedded systems. They proposed an operation zone-based load-balancer that avoids too frequent unnecessary load balancing and consequently minimizes heavy overheads related to double lock migration, cache invalidation, and high synchronization cost. Their approach defers load balancing till the current utilization of each CPU is not seriously imbalanced. In their approach, three zones: cold, hot and warm, based on CPU utilization (low, high and medium), were created, and different policies were applied for different zones [14].

In their paper, Tan et al. present an adaptive load balancing strategy. The adaptive load balancer triggers tasks migration based on the tasks to processing core ratio, as well as when a processing core becomes idle. The authors utilize LinSched, a Linux operating system scheduler simulator, to analyze the no. of task migrations. Results from the simulation show that unnecessary task migrations were eliminated, and at the same time, the load balance was maintained effectively, as compared to the default strategy used by Linux. The overheads introduced by the adaptive load balancer had a negligible effect on the scalability, and it was concluded that it does not introduce overheads [24].

There are several challenges Linux must address to improve the performance of NUMA systems. In their work, Focht et al. discuss these challenges, which include: localization of memory references, I/O locality, scheduling of processes on the parent node etc. A NUMA system can achieve better performance by keeping memory access to the closer physical memory. For example, processors benefit by accessing memory on the same node or nearer nodes. In [7], the authors describe how Linux addresses the above mentioned NUMA challenges and what are the gap areas. Lameter et al. in [2] have discussed the ways of minimizing the process migrations.

Chiang et al. in [3] [28] [30] have suggested the algorithms for improving the performance when inter-node process migration takes place. Contributions of Pilla et al. [21], Pusukuri et al. [22] and Khawatreh et al. [25] are also noteworthy.

Many other researchers have also suggested approaches for improving the Linux load balancer’s efficiency and thereby the performance of NUMA systems. The commonality in their work is the avoidance of unnecessary

load balancing and process migration attempts so as to optimize the system performance. However, despite a lot of work done in this direction, the scope of improvement always remains looking to the complexities of the load balancing mechanism and continuous evolution of new NUMA architectures. Novel ideas need to be generated and implemented to minimize the overheads of load balancing operations. Researchers need to think in an unconventional way also, for example, to make a trade-off between a perfectly balanced system (which may exhibit large overheads) and the slightly imbalanced system (which may exhibit better performance).

The work presented in this paper will be a significant contribution and supplement the efforts of the researchers towards the performance enhancement of NUMA systems by designing state-of-the-art load balancers.

IV. PROPOSED CONSERVATIVE LOAD BALANCING ALGORITHM

In order to improve the load balancer's performance, unnecessary overheads of load balancing attempts, process migration, and memory/cache access need to be avoided. Based on this key concept, a load balancer has been designed, which refrains from load balancing when these overheads are likely to degrade the performance. It carries out the load balancing operations in a restrictive or conservative manner and is thus named Conservative Load Balancer.

The proposed load balancer incorporates the following three conservative policies and performs the load balancing accordingly:

A. Restrict the load balancing within the 'Load Balancing Zone.'

As per the analysis of the present Linux load balancer, it was noted that all load balancing attempts and process migrations are not useful for performance improvement; in fact, some of them may also degrade the performance. Therefore, load balancing attempts and process migrations should be made in those conditions only wherein the performance is likely to improve. However, such predictions are difficult to be made, and hence attempts may be made to reduce the overheads of load balancing by avoiding it in those cases where the overheads will mostly be very high. A thorough analysis of the Linux load balancer revealed (as presented in the preceding sections) that performing the load balancing at higher levels of sched domain hierarchies may result in large overheads due to higher memory latencies and large no. of cache-misses.

The proposed conservative load balancer, therefore, restricts the scope of load balancing within a specific zone, named *Load Balancing (LB) Zone*, which, for a particular processor, comprises its parent node and the nodes in nearby memory access levels or sched domains. Load balancing is not attempted in the *No-Load Balancing Zone*, which comprises the nodes in the remaining sched domains.

More specifically, for no. of sched domain levels four or more, the load balancing zone comprises of all the nodes in the sched domains up to level S, where,

S = ((total no. of sched domains)/2 + 1).

For no. of sched domain levels two or three, the load balancing zone comprises all the nodes of all the sched domains. For example, for any particular Node (say Node N0), the load balancing zone will comprise of the Nodes at II, III and IV levels of sched domain hierarchy for a NUMA system with six levels of sched domains. Load balancer executing on a processor in N0 will balance its load against these Nodes only. Thus load balancing activities for the Nodes at V and VI levels are avoided, which in turn result in saving time and improving the performance.

Figure 3 depicts the Load Balancing Zone and No Load Balancing Zone for Node N0 for a NUMA system with six memory access levels and 16 Nodes.

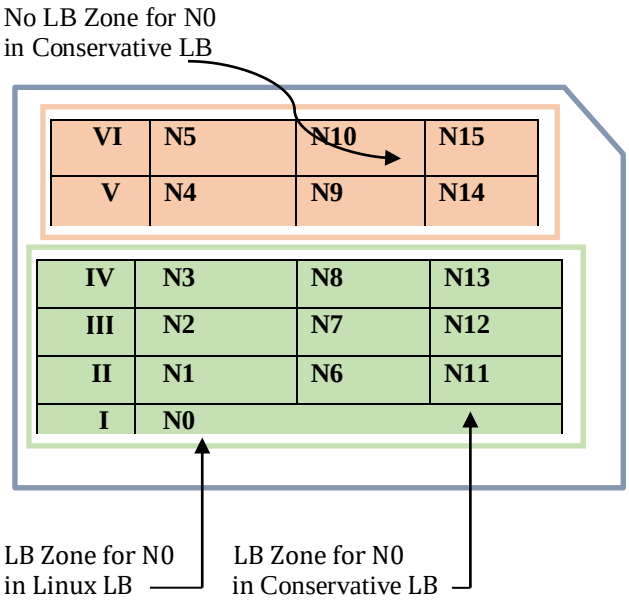

Fig. 3 LB Zone and No LB Zone in conservative load balancing and linux load balancing, for a node N0

The implementation of a Zone-based load balancing policy has been done as-

a) Hard policy

In this policy, strictly no balancing is done out of the load balancing zone.

b) Soft or Hybrid policy

Not performing load balancing at higher levels of sched domains will normally not result in any load imbalance or performance degradation in a highly dynamic process environment with a large no. of processes. However, to take care of any intolerable load imbalance which may occur in some cases, the Zone based load balancing policy is also implemented as a soft policy, which is a trade-off between the proposed hard policy and

the current Linux load balancing policy. As per this policy, loads of the nodes/processors of the *No load balancing zone* are also examined for detecting the load imbalance, if any. However, the processes from these nodes/processors are migrated only if their load is more than 150 % of the current processor's load. To implement this policy, the default load threshold is increased to 50 % for the processors which are out of the load balancing zone.

B. No Migration of Processes from No Load Balancing Zone

The proposed load balancer operates within the *Load Balancing Zone* of a processor on which it is currently executing. Within the zone, when it finds a load imbalance between the current processor and some other processor, it has to migrate the processes from that processor. However, the migration of any process which is originated on any node pertaining to the *No load balancing zone* for that process is avoided. In fact, the selection policy checks the *distance* between the current node and parent node of that process as well as the *distance* between the parent node and the node on which the process is proposed to be migrated (on which the load balancer is executing), as follows:

distance1 = *distance* between the current node and parent node of the process.
distance2 = *distance* between the current node and the proposed new node for the process.

If distance2 > distance1, the process is excluded from the list of the processes to be migrated. This modified process Selection Policy, therefore, results in fewer overheads related to memory and cache access.

C. No Migration of 'Aged' Processes

The load balancer employs the concept of *Ageing*, wherein it labels those processes which have been migrated a very large no. of times as *Aged* Processes and subsequently does not allow any further migration of such processes. A track of the migration history of each process is kept, and a counter associated with each process is incremented every time the process is migrated. After this counter crosses a particular value, the process is labelled as an *Aged* process.

Once an *ageing* process is frozen on its current node, its memory pages can be migrated to that node, minimizing the memory access overheads, apart from making the migration and cache-miss overheads zero for that process.

Following is the formal description of the Conservative Load Balancing Algorithm. The code given is for periodic load balancing. *Idle* and *Initial load balancing* is done in the usual manner.

Algorithm 1: Conservative Load Balancing

for all Nodes of the system N=1 to n and all processors P=1 to p of each Node, carry out the following steps:

```
1.  {
2.    max_sched_domains = no. of scheduling domains
      (Memory Access Levels) in this NUMA system;
3.    if the hard implementation is invoked, then
            //as per the setting in the kernel, either hard
              or soft implementation will be invoked.
4.   {
5.      hard_conser = TRUE;
6.      if (max_sched_domains >= 4) then
7.         sched_domains_in_LB_zone =
           (max_sched_domains/2 ) + 1  ;
8.    }
9.    else
10.    {
11.      soft_conser = TRUE;
12.      sched_domains_in_LB_zone =
         max_sched_domains;
13.     }
14.  for MAL=1 to sched_domains_in_LB_zone  do
15.  {
16.    if (MAL==1) then
17.    {
18.      processor_performing_LB = PP
         // PP is the idle processor or the first processor
19.      find the load of all processors of curr_node,
         except the  processor_performing_LB;
20.      find the busiest processor;
                // processor having highest load
21.    }
22.    else   // if MAL is > 1
23.    {
24.      processor_performing_LB=PP ;
25.      find the busiest scheduling group out of all the
         scheduling groups (all nodes) at memory
         access level MAL;
26.      find the  busiest processor of  the  busiest node
         (scheduling group with  highest load);
27.      }            //end of if statement at step no. 16
28.      LB_processor_load = load of
             processor_performing_LB;
29.      target_processor_load = load of the busiest
                                  processor;
30.      if (LB_processor_load  <
             target_processor_load) then
                  // compare the load of the
                    processor_performing_LB  with  that
                    of the busiest processor
31.      {
32.        if (soft_conser == TRUE) .and. (MAL >
             (max_sched_domains/2 ) + 1  ) then
                load threshold = load threshold * 2;
33.        obtain lock on target_processor;
              // busiest processor is the  target processor
34.        obtain lock on processor_performing_LB;
35.        select appropriate no. of processes for
           migration;
```

```
36.     if ((hard_conser == TRUE) .and. (any
        process selected for migration belongs to
        scheduling domain outside the load
        balancing zone of
        processor_performing_LB)) then
37.         exclude all such processes from the list of
            processes to be migrated and instead
            select other processes, if available;
38.     endif;
39.     if any process selected for migration is aged
        process then
40.         exclude all such processes from the list
            of processes to be migrated and instead
            select other processes, if available;
41.     endif;
42.     migrate the finally selected processes from
        busiest processor to
        processor_performing_LB;
            // pull the processes/threads from the
               busiest processor till the load of the
               two processors remain imbalanced, ie.
               dequeue the selected process from the
               target processor and enqueue on the
              processor_performing_LB;
43.     release lock on processor_performing_LB;
44.     release lock on target_processor;
45.   }          //end of if statement at step no. 30
46.  MAL=MAL+1;
47.  }           // end of for loop at step no. 14
48. }            // end of Algorithm
```

V. SIMULATION AND RESULTS

To evaluate the performance of the Conservative Load Balancing Algorithm, experimentation was done using a simulator of NUMA Multiprocessor systems under Linux [19], modified by incorporating the proposed algorithm into it.

The experimentation was done for different types of NUMA Systems-

(i) **S1**: No. of Nodes=16, No. of Processors per Node=2, No. of Memory Access Levels=6
(ii) **S2**: No. of Nodes=16, No. of Processors per Node=4, No. of Memory Access Levels=6
(iii) **S3**: No. of Nodes=32, No. of Processors per Node=2, No. of Memory Access Levels=6.

For each system, a variety of workloads (W1, W2, W3) were generated.

A. Results

a) Turn Around Time and Performance Gain

The results of simulation in terms of Av. Turn Around Time (ms) and Performance Gain (%)are given in Tables 3 to 5 and are also depicted in the corresponding graphs given after the respective Tables (Workload Characteristics are specified as W1, W2, W3 in each Table and Graph).

Table 3. Turn around time of processes and performance gain for conservative load balancing algorithm vs linux load balancing algorithm for NUMA system S1

No. of processes	**W1-** Process type: CPU bound; Execu. time: 200 ms; Arrival: same time			**W2-** Process type: CPU- bound; Execu. time: 200 ms; Arrival: random			**W3-** Process type: CPU-bound; Execu. time: 300 ms; Arrival: random		
	Linux Algo.	***Conser-vative Algo.***	***Perf. Gain (%)***	***Linux Algo.***	***Conservative Algo.***	***Perf. Gain (%)***	***Linux Algo.***	***Conser-vative Algo.***	***Perf. Gain (%)***
50	376	361	3.98	459	448	2.40	690	645	6.52
100	633	588	7.11	620	577	6.94	1010	909	10.00
200	1263	1174	7.05	993	896	9.77	1750	1581	9.66
300	1799	1620	9.95	1449	1303	10.08	2470	2292	7.21
400	2402	2237	6.87	1815	1613	11.13	3047	2805	7.94

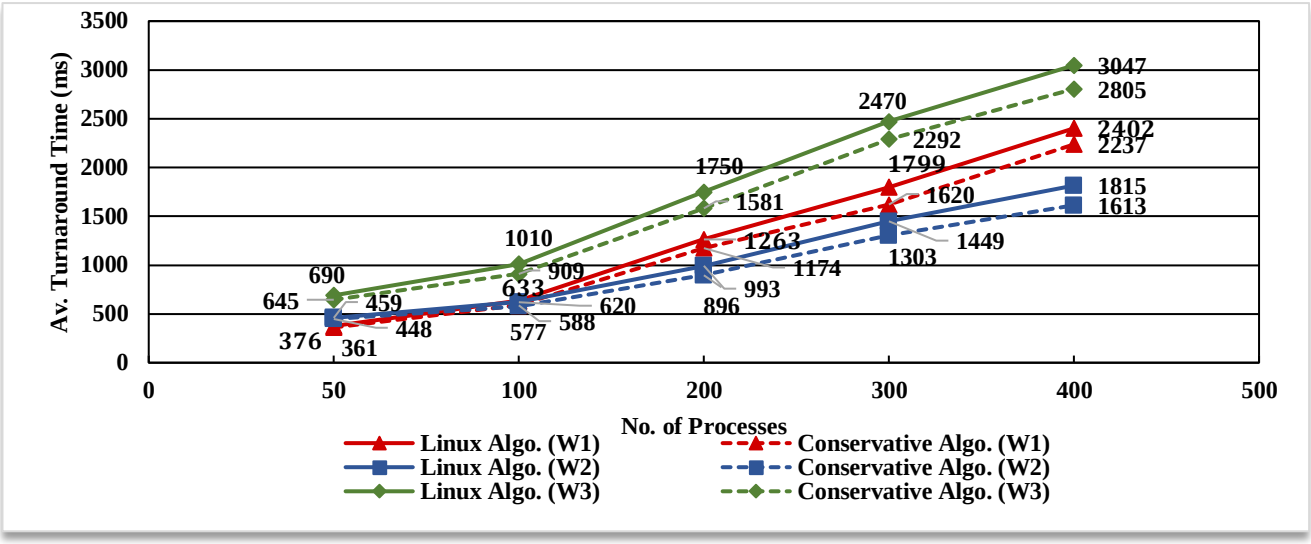

Fig. 4 Turn around time of processes for conservative load balancing algorithm vs linux load balancing algorithm for NUMA System S1

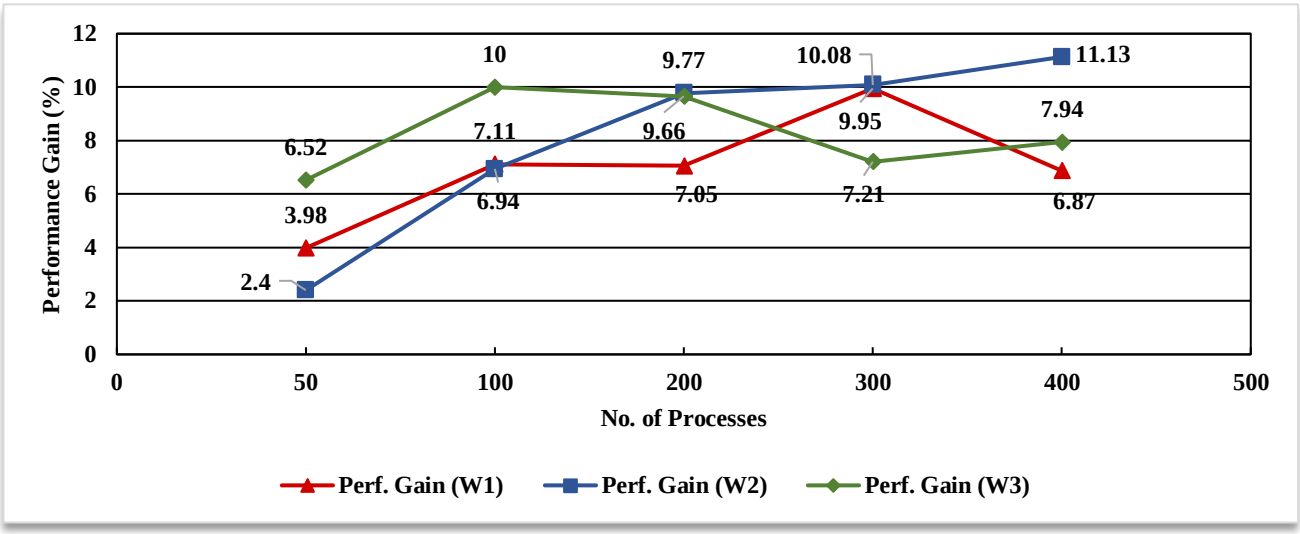

Fig. 5 Performance Gain in Conservative Load Balancing over Linux Load Balancing for NUMA System S1

Table 4. Turn around time of processes and performance gain for conservative load balancing algorithm vs linux load balancing algorithm for NUMA system S2

No. of processes	**W1-** Process type: CPU- bound; Execu. time:300 ms; Arrival: random			**W2-** Process type: CPU bound; Execu. time: 300 ms; Arrival: almost same time			**W3-** Process type: Mix of CPU & IO-bound; Execu. time: varying (50-400 ms); Arrival: almost same time		
	Linux Algo.	***Conser-vative Algo.***	***Perf. Gain (%)***	***Linux Algo.***	***Conserva tive Algo.***	***Perf. Gain (%)***	***Linux Algo.***	***Conser-vative Algo.***	***Perf. Gain (%)***
50	875	860	1.71	560	534	4.64	419	404	3.58
100	977	928	5.02	823	762	7.41	552	510	7.61
150	1202	1098	8.65	1080	979	9.35	691	623	9.84
200	1545	1366	11.59	1277	1155	9.55	820	755	7.93
250	1639	1470	10.31	1627	1453	10.69	1062	936	11.86
350	2026	1812	10.56	2206	2025	8.20	1363	1243	8.80

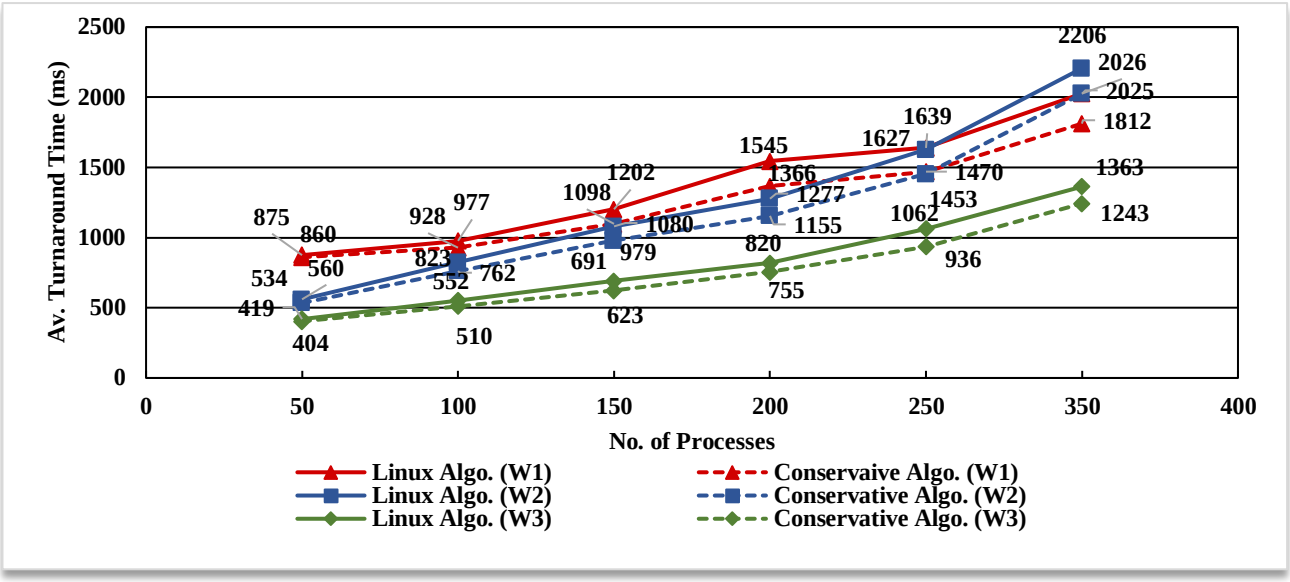

Fig. 6 Turn around time of processes for conservative load balancing algorithm vs linux load balancing algorithm for NUMA system S2

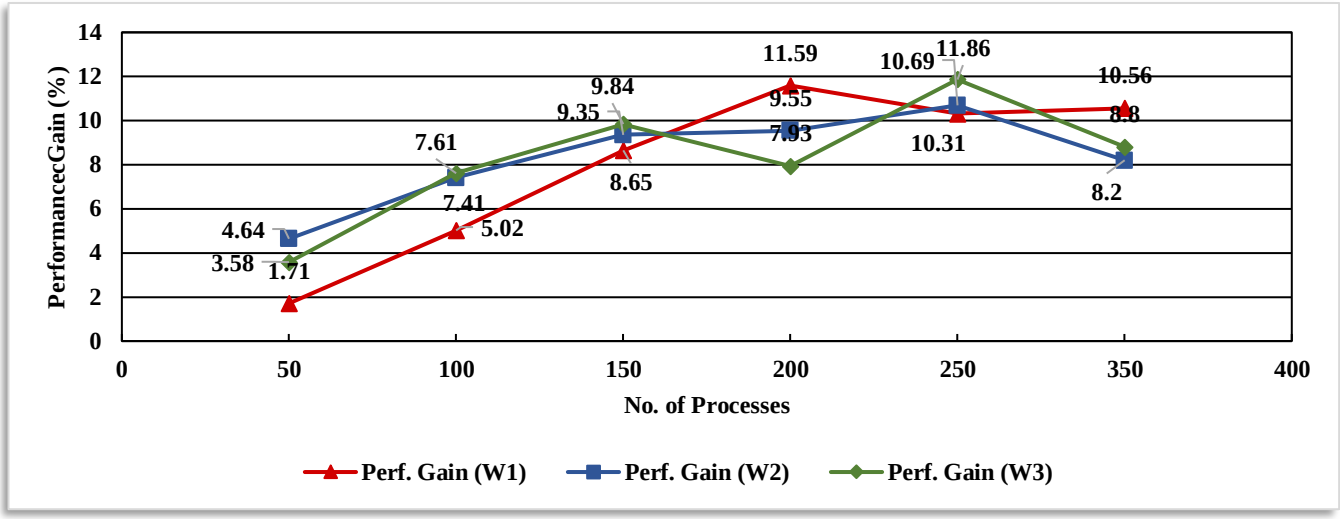

Fig. 7 Performance gain in conservative load balancing over linux load balancing for NUMA system S2

Table 5. Turn around time of processes and performance gain for conservative load balancing algorithm vs linux load balancing algorithm for NUMA system S3

No. of processes	**W1-** Process type: CPU bound; Execu. time:300 ms; Arrival: random			**W2-** Process type: CPU bound; Execu. time: varying (100-500 ms); Arrival: almost same time			**W3-** Process type: Mix of CPU & IO-bound; Execu. time: 400 ms; Arrival: random		
	Linux Algo.	***Conservative Algo.***	***Perf. Gain (%)***	***Linux Algo.***	***Conser-vative Algo.***	***Perf. Gain (%)***	***Linux Algo.***	***Conser-vative Algo.***	***Perf. Gain (%)***
100	1061	966	8.95	679	610	10.16	1234	1123	09.00
200	1564	1414	9.59	942	848	9.98	1918	1737	09.44
300	2038	1883	7.61	1592	1422	10.68	2472	2225	09.99
400	2676	2453	8.33	2164	1987	8.18	3376	3028	10.31

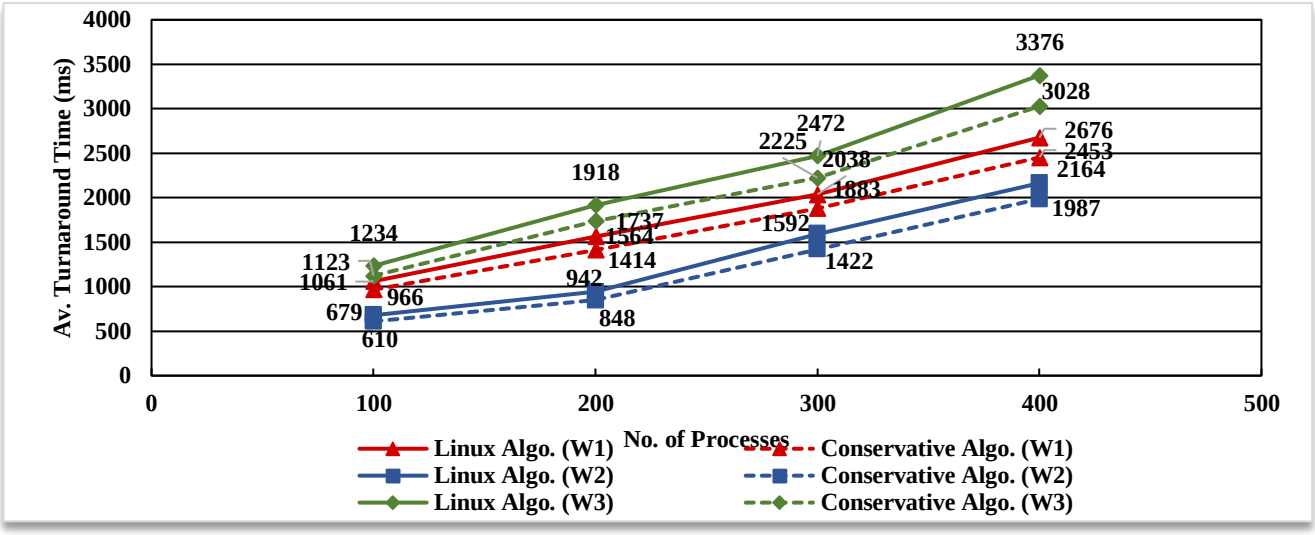

Fig. 8 Turn around time of processes for conservative load balancing algorithm vs linux load balancing algorithm for NUMA system S3

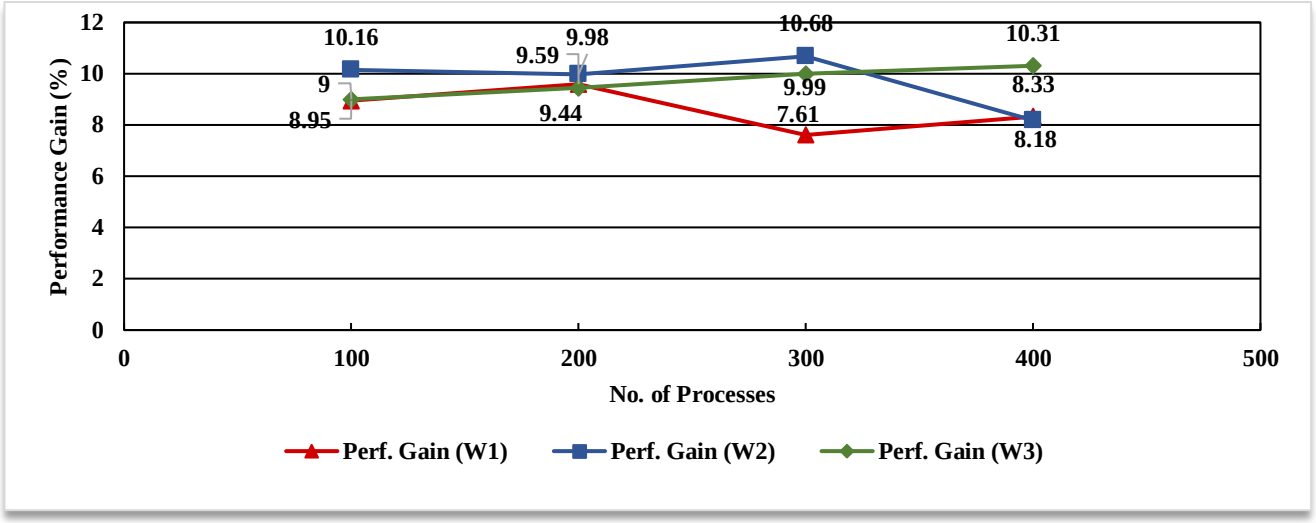

Fig. 9 Performance gain in conservative load balancing over linux load balancing for NUMA system S3

B. Traces of Few Processes Depicting their Migration Zones

Traces of a few sample processes, out of 300 processes executed, were obtained from the simulator and are illustrated below. From these traces, it can be seen that processes are normally not migrated out of their *Load Balancing Zone* (only a few processes are migrated to *No Load Balancing Zone* as shown in bold):

Process-67: P5.wait(124) - P5.run(95)- IO(5)-P5.wait(34)- P2.wait(266) - P2.run(79)- IO(21)-P2.wait(33)- P9.wait(128)- P28.wait(528)- P28.run(12).

Process-184: P13.wait(24)- P24.wait(73)- P26.wait(125)- P7.wait(149)- **P2.wait(152) - P2.run(100) -P2.wait(95) - P2.run(82)- IO(18)-P2.wait(0)- P2.run(23).**

Process-166: P8.wait(304) - P8.run(100) -P8.wait(62)- P5.wait(155)- P4.wait(106) - P4.run(78)- IO(22)- P4.wait(197)- P4.run(17).

Process-60: P21.wait(184) - P21.run(77)- IO(23)- P21.wait(215)- **P18.wait(687) - P18.run(86)- IO(14)- P18.wait(63)-** P14.wait(152)- P14.run(18).

Process-155:P15.wait(19)- P28.wait(273)-P28.run(100)- P28.wait(519)- P29.wait(76)-P29.run(100)-P29.wait(0)- P29.run(30).

Process-185:P19.wait(311)- P19.run(100)-P19.wait(111)- P4.wait(142)-P4.run(100)-P4.wait(137)- P4.run(12).

Process-36: P29.wait(82)- P12.wait(61)- P23.wait(43)- P6.wait(-18) - P6.run(100) -P6.wait(387) - P6.run(100) - P6.wait(881)- P6.run(20).

Process-164: P1.wait(157)- P12.wait(206) - P12.run(87)- IO(13)-P12.wait(8)- P14.wait(259) - P14.run(80)- IO(20)- P14.wait(21)- **P9.wait(117)**- P2.wait(38)- P2.run(23).

Process-171: P21.wait(11)- **P28.wait(379) - P28.run(58)-IO(42)-P28.wait(17)- P29.wait(186) - P29.run(67)-IO(33)-P29.wait(36)-** P7.wait(63)- P17.wait(119)-P17.run(40).

Process-241: P10.wait(21)- P19.wait(511) - P19.run(69)-IO(31)-P19.wait(261)- P18.wait(301)- P27.wait(174) - P27.run(100) -P27.wait(668)- P27.run(35).

Process-280: P22.wait(126)- P8.wait(198)- P13.wait(166)-P15.wait(100)- P4.wait(218) - P4.run(84)- IO(16)-P4.wait(31)- P0.wait(48) - P0.run(100) -P0.wait(0)-P0.run(7).

Process-214: P5.wait(15)- P22.wait(529) - P22.run(66)-IO(34)-P22.wait(214)- P8.wait(185)- P13.wait(160) - P13.run(72)- IO(28)-P13.wait(61)- P12.wait(72)-P12.run(28).

B. Observations and Discussion on Results

It is evident from the experimental results that the Conservative Load Balancing Algorithm outperforms the Linux load balancing algorithm for various NUMA systems having different architectures and exhibits the better average TAT in the range of 7-12%. The achieved performance gain is attributed to the reduced load balancing overheads.

It is further observed that-

a) Majority of processes remain within their Load Balancing Zone only; even many processes are completely executed on the originating processor/node. Very few processes are migrated to far nodes when the load balancing policy is implemented as a soft conservative policy. This is evident from the traces of a few sample processes (shown in sub-section A.2 of this section) after performing load balancing through Conservative Load Balancer. Non-migration of most of the processes too far nodes results in fewer overheads and, in turn, more performance gain.

b) The performance gain is almost in the same range for the three NUMA systems for which experimentation was done. This is because all the systems have the same no. of memory access levels. For the systems with more levels, higher gains will be achieved.

c) Relatively small performance gain (in the range 2-4 %) observed in a few cases with a small no. of processes is due to a good process to processor ratio. In such cases, there will be relatively fewer overheads of load balancing in the Linux algorithm and hence less performance gain in Conservative Load Balancing.

d) There is variation in performance gain for different sets of processes for the same system. This is due to differences in the characteristics of the workload, like, no. of processes, arrival time of the processes, different no. Of computation and input-output instructions in a process etc.

VI. CONCLUSION

The major factors responsible for the non-optimum performance or performance degradation of any load balancing algorithm are the overheads of load balancing and process migration. The overheads are aggravated for the NUMA systems with large no. of memory access levels. Thus, avoidance of unnecessary load balancing operations is a key factor for the enhancement of any load balancer.

In this research, the approach followed by Linux for load balancing was investigated and analyzed, and a Conservative Load Balancer was proposed to achieve better performance. On the basis of simulation results, it can be concluded that the proposed load balancer has successfully addressed the issue of unnecessary load balancing overheads incurred in Linux and improved the performance very significantly. The work presented will supplement the endeavours of the researchers attempting to design efficient load balancing algorithms for upcoming NUMA multiprocessor and multicore systems.

.

REFERENCES

[1] Martin J. Bligh, M. Dobson, D. Hart, and G. Hu Lzenga, Linux on NUMA Systems, Linux Symposium, (2004).

[2] Christoph Lameter, Process Scheduling on 1024 Processors, in Proc. Gelato ICE 2007: Itanium Conference & Expo, San Jose, California, (2007).

[3] Mei-Ling Chiang, Shu-Wei Tu, Wei-Lun Su, and Chen-Wei Lin, Enhancing Inter-Node Process Migration for Load Balancing on Linux-Based NUMA Multicore Systems, in Proc. IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), (2018).

[4] Priyesh Kanungo, Contributions in Dynamic Load Balancing Techniques for Distributed Computing Environment, Ph.D. Thesis, IET-DAVV, (2007).

[5] M. Correa, R. Chanin, A. Sales, R. Scheer, and A. Zorzo, Multilevel Load Balancing in NUMA Computers, Technical Report No. 49, PPGCC-FACIN-PUCRS, Brazil, (2005).

[6] Dejan S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, S. Zhou, Process Migration, ACM Computing Survey, 32(3) (2000) 241-299.

[7] Erich Focht, Mathew Dobson, Patricia Gaughen, and Michael Hohnbaum, Linux Support for NUMA Hardware, Linux Symposium, (2003).

[8] S. Hofmeyr, C. Iancu, and F. Blagojevi, Load Balancing on Speed, in Proc. 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10), New York, USA, (2010) 147-158.

[9] Weiwei Jia, How does load balancing work inside of operating systems, Linux as an example, Available:https://www.systutorials.com/load-balancing-work-internal-operating-systems (accessed June 9, 2020).

[10] M. Tim Jones, Inside the Linux Scheduler, Available: http://www.ibm.com, (accessed Jan. 27, 2015).

[11] Sandeep Sharma, Sarabjit Singh, and Meenakshi Sharma, Performance Analysis of Load Balancing Algorithms, in Proc. World Academy of Science, Engineering and Technology, 28 (2008).

[12] Linux Kernel Documentation [Online]. Available: https://www.kernel.org/doc/html/latest/ (accessed Jan. 2021).

[13] Adam Wynne, Load Balancing in Distributed Operating Systems, Technical Report, Western Washington Univ., (2005).

[14] Geunsik Lim, Changwoo Min, and Younglk Eom, Load-Balancing for Improving User Responsiveness on Multicore Embedded Systems, Linux Symposium, (2012).

[15] Ye Liu, Shinpei Kato, and Masato Edahiro, Optimization of the Load Balancing Policy for Tiled Many-Core Processors, IEEE Access Journal, (2018).

[16] Robert Love, Linux Kernel Development, Novell Press, 2nd edition, (2005).
[17] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Qu´ema, and Alexandra Fedorova, The Linux Scheduler: a Decade of Wasted Cores, in Proc. EuroSys '16, London, UK, (2016).
[18] N. Padhy, A. Panda, and S.P. Patro, A Cyclic Scheduling for Load Balancing on Linux in Multi-core Architecture, in Proc. Third International Conference on Smart Computing and Informatics, (2019) 369-378.
[19] Shreelekha Pandey, Simulator for Linux Scheduler and Load Balancer for NUMA Multiprocessor Architectures, M.E. Dissertation, S.G.S.I.T.S., (2009).
[20] Shreelekha Pandey, D.A. Mehta, Simulator of NUMA Multiprocessor Environment & Linux Load Balancing Scheduler, IJCEE, (2013).
[21] L. L. Pilla et al., A Hierarchical Approach for Load Balancing on Parallel Multi-Core Systems, in Proc. 41st Int. Conf. Parallel Processing (ICPP), (2012) 118–127.
[22] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan, Tumbler: An Effective Load Balancing Technique for MultiCPU Multicore Systems, ACM Transactions on Architecture and Code Optimization, Springer, Singapore, 160 (2015).
[23] Suresh Siddha, sched: new sched domain for representing multicore, Available: http://lwn.net/Articles/169277/ (accessed Feb. 2014).
[24] Ian K. T. Tan, Ian Chai, and Poo Kuan Hoong, An Adaptive Task-Core Ratio Load Balancing Strategy for Multi-core Processors, International Journal of Computer and Electrical Engineering, 3(5) (2011).
[25] Saleh A. Khawatreh, An Efficient Algorithm for Load Balancing in Multiprocessor Systems, International Journal of Advanced Computer Science and Applications (IJACSA), 9(3) (2018).
[26] Automatic NUMA Balancing, Available: https://documentation.suse.com/sles/15- SP1/html/SLES-all/cha-tuning-numactl.html, (accessed April 1, 2020).
[27] B. Mallikarjuna, D. Arun Kumar Reddy, and N Venkata Vinod Kumar, Green Computing: Efficient Energy Load balancing Technique in Cloud computing, International Journal of Computing Communications and Data Engineering, (2018).
[28] M.L. Chiang, W.L. Su, S.W. Tu, and Z.W. Lin, Memory-Aware Kernel Mechanism and Policies for Improving Inter-Node Load Balancing on NUMA Systems, in Software: Practice and Experience, John Wiley, NJ, USA, 49 (2019) 1485–1508.
[29] J. Chen, S. S. Banerjee, Z.T. Kalbarczyk, and R.K. Iyer, Machine Learning for Load Balancing in the Linux Kernel, in Proc. The 11th ACM SIGOPS Asia-Pacific Workshop on Systems, Tsukuba, Japan, (2020).
[30] Mei-Ling Chiang and Wei-Lun Su, Thread-Aware Mechanism to Enhance Inter-Node Load Balancing for Multithreaded Applications on NUMA Systems, Applied Sciences, Available: https://doi.org/10.3390/app11146486, (accessed Nov. 2021).