International Journal of Engineering Trends and Technology
ISSN: 2231–5381 / https://doi.org/10.14445/22315381/IJETT-V73I2P102
Volume 73 Issue 2, 9-21, February 2025

Original Article

Origin Aware Dynamic Load Balancing Algorithm for Performance Enhancement of NUMA Multiprocessor Systems

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

[1]*Shri G. S. Institute of Technology & Sc., Indore, India.*
[2]*School of Computer Science, Devi Ahilya Vishwavidyalaya, Indore, India.*

[1]*Corresponding Author : mehta_da@hotmail.com*

Received: 02 August 2024 Revised: 11 December 2024 Accepted: 17 December 2024 Published: 21 February 2025

Abstract - *The process selection policy of the linux load balancer pays no attention to the origin of the processes while selecting them for migration in NUMA Multiprocessor Systems. Consequently, the migrated processes may experience large memory latencies, and the load balancer may degrade the system performance, particularly when the number of memory access levels is large. This paper proposes a novel load balancing algorithm for NUMA Multiprocessors that attempts to keep the processes on or near their originating nodes and thereby reduces the memory access overheads to zero or minimum, resulting in significant performance gain (ranging from 7 to 23% for various NUMA systems) over the existing load balancer.*

Keywords - *Dynamic load balancing, Load balancer, NUMA, Scheduling domain, Process migration, Memory Access Level, Memory access overhead.*

1. Introduction

Non Uniform Memory Access (NUMA) architecture is commonly used for designing modern multiprocessor and multicore systems. A NUMA Multiprocessor/ Multicore system is designed in terms of Nodes. Each node contains a set of processors and a portion of the main memory placed on a common bus. A high-speed interconnection network connects the nodes with one another. Memory in a specific node is at a distance (referred to as latency) from the processors of other nodes, causing non-uniform access times of on-Node and off-Node memories [1, 13]. Figure 1 represents a NUMA multiprocessor system with two Memory Access Levels (MALs). It is said to have two MALs because of two different memory latencies: When an access of memory is made by a processor in its own node and when an access is made in a different node [3].

1.1. Dynamic Load Balancing

Linux incorporates a Dynamic Load Balancer in its scheduler to keep the load of the processors balanced in NUMA Multiprocessors. This load balancer uses a data structure called ‘sched domain’ to organize the processors in a hierarchy that imitates the physical hardware. It consists of a group of processors sharing the properties and scheduling policies [16]. Figure 2 shows the scheduling domain hierarchy for the NUMA system depicted in Figure 1. The lowest level scheduling domains are named CPU/Core domains, each of which comprises all the processors of a specific node and points to a higher domain, the node domain, comprising of this node and all the nodes that are located at some distance from it [3]. The scheduling domain hierarchy defines the scope/extent of load balancing for each processor. The Load balancer performs the load balancing when triggered in the following conditions [11, 15]:

- Periodically at regular time intervals: The complete scheduling domain hierarchy is periodically traversed, beginning at the scheduling domain of the current processor, and a balancing operation is initiated. At each level, the most loaded processor of the most loaded scheduling group is looked for, and migration of tasks takes place from that processor to the current processor if the busiest processor’s load is greater than the current processor’s load according to the load threshold, which is normally 25%.

- When a new task is created or a task wakes- up: This task is allocated the least loaded processor of the least loaded scheduling group (node) in its current domain.

- Whenever a processor goes idle, *idle load balancing* is carried out by the idle processor, and tasks are migrated from the busiest processor of the busiest scheduling group in its current domain to this processor.

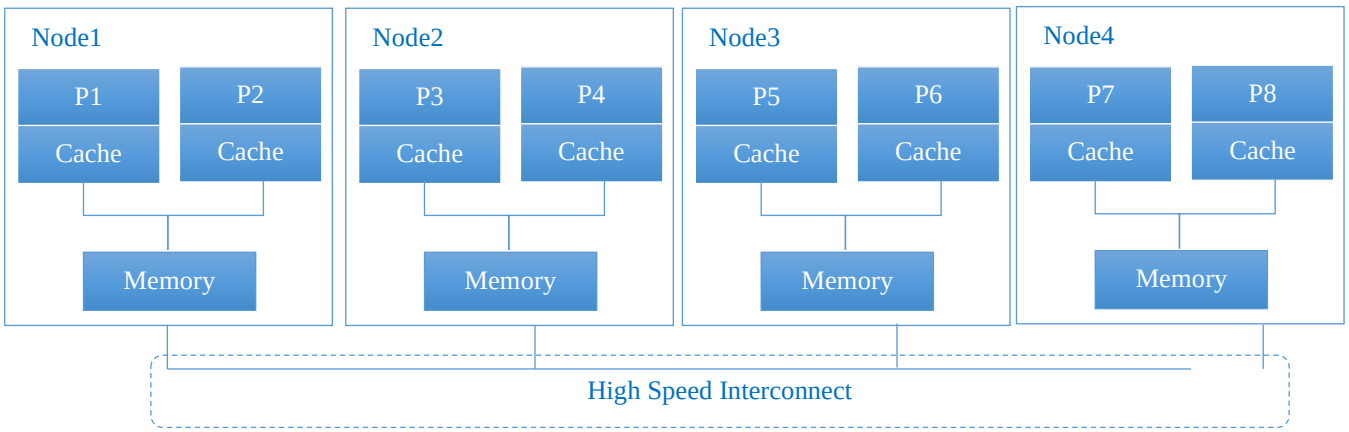

Fig. 1 NUMA multiprocessor system having two memory access levels (P1, P2… are CPUs)

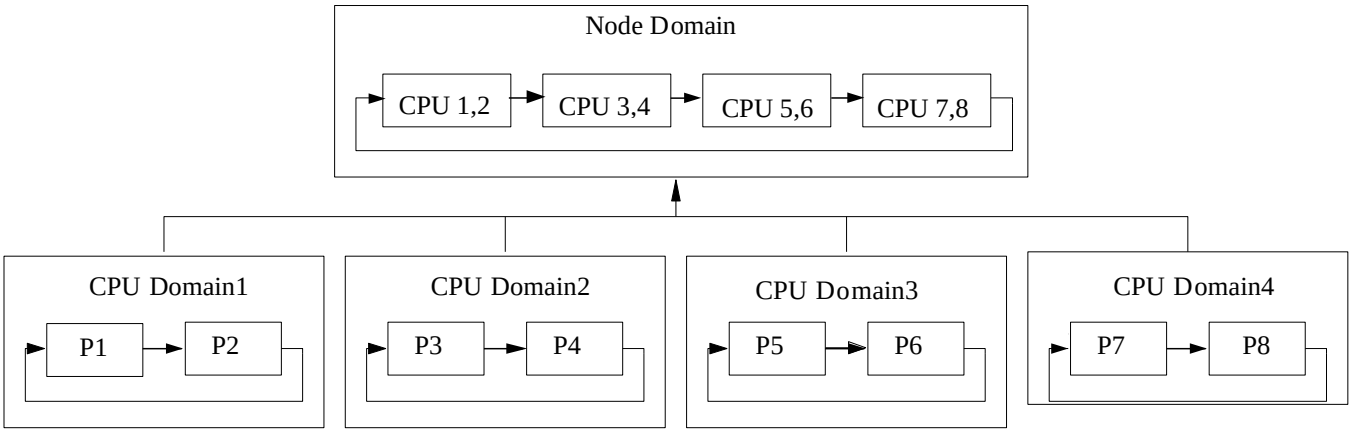

Fig. 2 Scheduling domain hierarchy for NUMA multiprocessor system having two memory access levels

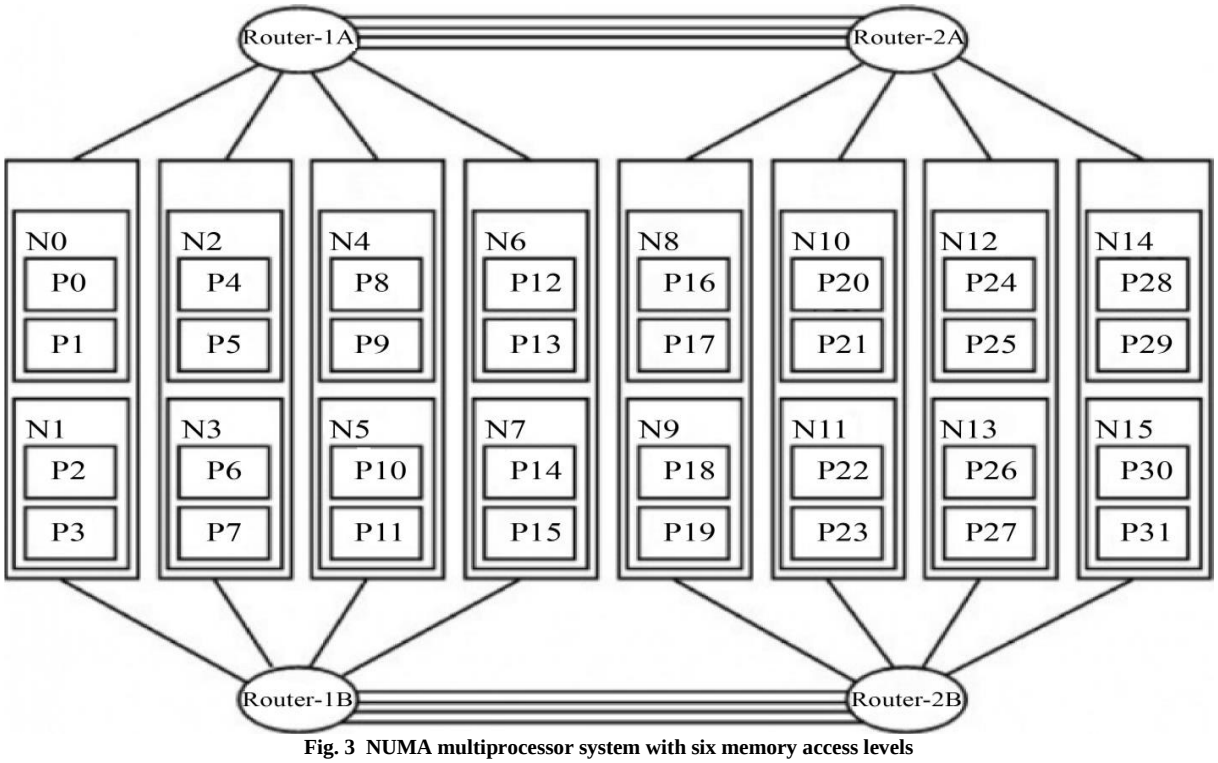

Fig. 3 NUMA multiprocessor system with six memory access levels

The preceding description makes it clear that dynamic load balancing involves a large number of process migrations, resulting in significant memory access overheads due to the penalty incurred when data is accessed from a remote memory instead of local memory. However, the process selection policy of the load balancer does not keep the origin of the processes in view while selecting them for migration. Consequently, the linux load balancer results in large memory latencies, especially when the number of memory access levels are higher.

2. Process Migration in Linux: An Analysis

In order to understand the process migration policy of the linux load balancer and to explore the possibility of its improvement, the load balancing operation for a NUMA system with an architecture similar to the system shown in Figure 3, with 6 Memory Access Levels, 16 Nodes and 2 Processors per Node [3], is explained here.

For this example system, Table 1 shows (partially for a few Nodes) the relative distance (memory latency) between the nodes. In the table, the value of a position Pi,j denotes the distance from node i to node j. The distance from a node to itself is called SMP distance, and its default value is 1x. All other distance values are relative to SMP distances.

For instance, the distance from node N0 to node N1 is 2x, meaning that a processor in N0 accesses a memory area in N1 two times slower as compared to the memory area in N0 [3]. (N0, N1, N2 … N15 are Nodes; P0, P1 … P31 are Processors) Figure 4 depicts the nodes at various sched domain hierarchy levels for a particular node N0.

As shown, for node N0, nodes N1, N6, N11 are II-level nodes; N2, N7, and N12 are III-level nodes, and so on. The above information is used by a load balancer while selecting the processes for migration.

Table 1. Memory latencies of different nodes from a particular node

	N0	**N1**	**N2**	**N3**	**N4**	**N5**	**N6**	**N7**	**N8**	**N9**	**N10**	**N11**	**N12**	**N13**	**N14**	**N15**
N0	1x	2x	3x	4x	5x	6x	2x	3x	4x	5x	6x	2x	3x	4x	5x	**6x**
N1	2x	1x	2x	3x	4x	5x	6x	2x	3x	4x	5x	6x	2x	3x	4x	**5x**
N2	3x	2x	1x	6x	2x	3x	4x	5x	6x	2x	3x	4x	5x	6x	2x	**3x**
N3	**4x**	**3x**	**6x**	**1x**	**4x**	**5x**	**6x**	**2x**	**3x**	**4x**	**5x**	**6x**	**2x**	**3x**	**4x**	**5x**

VI	**N5**		**N10**		**N15**	
	P10	P11	P20	P21	P30	P31
V	**N4**		**N9**		**N14**	
	P8	P9	P18	P19	P28	P29
IV	**N3**		**N8**		**N13**	
	P6	P7	P16	P17	P26	P27
III	**N2**		**N7**		**N12**	
	P4	P5	P14	P15	P24	P25
II	**N1**		**N6**		**N11**	
	P2	P3	P12	P13	P22	P23
I	**N0**					
	P0		P1			

Fig. 4 Nodes at different sched domain hierarchy levels for node N0
(N0, N1 … are Nodes and P0, P1 …. are Processors in the different Nodes)

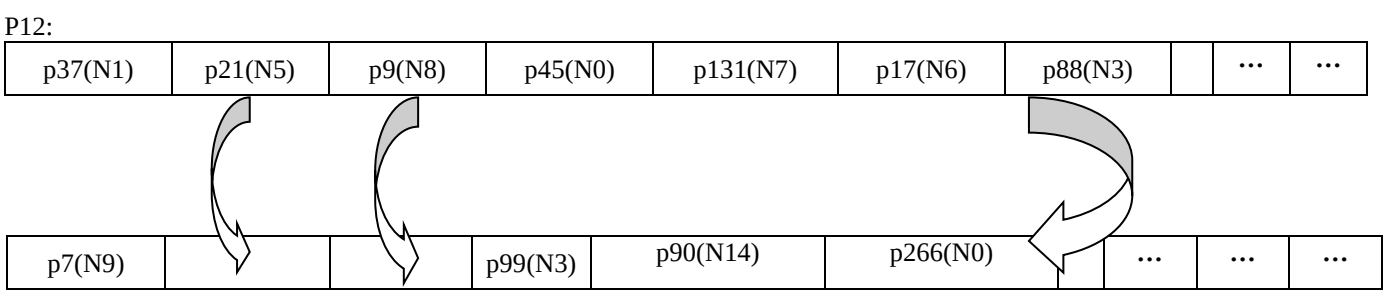

Fig. 5 Depiction of process migration in linux load balancing
(P1, P12 are runqueues of Processors P1 & P12; pi (Nj) is the process having pid=i and parent Node Ni)

For a load balancing cycle, the load balancer executing on a processor (say P1, in Node N0) performs the load balancing in N0, then finds the busiest processor in the next level Nodes N1, N6, N11 (suppose it is P12, in N6), and finding the load imbalance between P1 and P12, pulls the processes p21, p9, and p88 which originally belong to distant nodes with reference to current node, even though processes p37, p45 and p17 belonging to N0 or the nearby nodes, are present. Figure 5 depicts the runqueues of processors P1 and P12 and the process migration done to balance the load between these two processors.

In order to analyze the Linux load balancer's functioning, load balancing was performed for the NUMA system, for example, with various workloads consisting of different numbers of CPU-bound processes having varying execution times and random arrival. As per the simulation results given in Table 2, a large % of processes remained away from their originating Nodes/processors throughout their lifespan. In general, a large number of processes are migrated away, and no attempt has been made to send them back to their parent nodes. This policy of migration results in large overheads related to memory access and, consequently, in the degraded or non-optimum performance of the system. In a load-balancing scenario, the total memory access time of all the processes can be computed as:

Total Memory Access Time (TMAT) =

$$\sum_{i=1}^{n} \sum_{MAL=1}^{L} P_i\_nma_{MAL} * Tma_{MAL}$$

Where, n = No. of processes (P1 to Pn)
L = No. of Memory Access Levels (MAL=1 to L)
$P_i\_nma_{MAL}$ = No. of memory accesses done by process P_i at a particular Memory Access Level MAL
Tma_{MAL} = Time required for one memory access at Memory Access Level MAL

Table 2. No. of processes executed on the nodes distant from their parent nodes (for example, NUMA system and workload)

Total No. of Processes	Processes Executed on the Nodes Distant from Respective Parent Nodes	
	No. of such processes	% of such processes
100	46	46 %
200	96	48 %
300	129	43 %

It is clear from the above expression that to reduce the TMAT and thus the Av. TAT of the processes and number of accesses done by any process to remote memories should be minimized. The foregoing analysis of linux load balancing reveals the key point for performance enhancement; that is, attention should be paid to the originating Node of any process while balancing the load of the processors. The load balancer proposed in this paper considers this point to reduce memory access time.

3. Related Work

The key to performance enhancement of load balancing techniques is to avoid unnecessary process migrations and to minimize the memory access overheads if all the migrations are necessary. This section presents the work done by the researchers in order to minimize the memory and cache access overheads. Focht et al. [5] discuss measures for performance improvement, such as localizing the memory references and input/output, executing the processes on their originating nodes, etc. Khawatreh in [18] discuss the methods of reducing the process migrations to make the load balancer efficient. Kermia et al. [12], Pusukuri et al. [14] and Khawatreh et al. [17] have also made significant contributions in this direction. In a multithreaded environment, a load balancer should either avoid migration of threads among nodes or select the threads for migration so that memory access overheads are minimal.

A technique to avoid the migration of threads to remote nodes by limiting their movement within smaller zones is proposed by authors in [9]. Diener et al. propose in [10] a Kernel-based mechanism for thread and data mapping for improving memory affinity. Chiang et al. have also discussed the locality issue and designed the appropriate policies for selecting the threads for migration. Proper mapping of threads to cores and data access to their nodes significantly enhance the system performance; on the contrary, migration of inappropriate threads degrades it. Considering this important aspect related to selecting appropriate threads for migration, the authors presented two important policies. In the first one, a memory-aware kernel mechanism and inter-node thread migration policies are proposed to reduce remote memory accesses. Modifications in the load balancing approach of Linux for inter-node thread migration were made to track the memory usage of each thread on each node. Based on this information, the load balancer can select appropriate threads for the migration [19].

The second policy, the thread-aware selection policy, considers the distribution of threads on nodes for every thread group while migrating a thread for inter-node load balancing. The migrated threads must be selected effectively since the related operations run in the critical path of the scheduler. The experimental results show a performance improvement of approximately 11 % against the existing linux kernel [20]. The author in [4] describes the work on optimization of thread affinity and memory affinity for remote core locking synchronization in multithreaded programs for multicore systems. Another approach to minimize memory access overheads in NUMA systems is to use operating system page protection mechanisms to induce faults to find which thread accesses which pages to migrate the thread and its working set

of pages to the same node. However, many existing mechanisms do not fully fit the requirements of truly multithreaded applications with non-partitioned access to virtual pages. Thus, the fault of one thread may mask those of other threads on the same page, resulting in inaccuracy in estimating the working set of individual threads. To address this problem, a lightweight O/S support for linux, named 'multi-view address space', is proposed in [7]. Transferring the memory pages of the process to the same node to which the process has been migrated can be an alternative solution to bring down memory access overheads to zero. However, the migration of memory pages is a resource-intensive operation and has an impact on the system's overall performance. Therefore, an approach to migrate only the demanded pages, and not all the pages, is proposed in previous study.

The decision to migrate a process's memory is, therefore, not trivial, and many factors need to be taken into account before making a final decision [6]. In recent work, Chiang et al. also reached a similar conclusion that memory page migration, if not done very carefully, may degrade the performance. In [2], the authors point out that, though the current linux kernel transfers the referenced memory page of a process to the node it is presently executing, on the occurrence of a page fault, this migration incurs additional memory access overheads because of the costly page fault handling and page migration operations. In another work, Barrera et al. suggested a method to reduce the memory page migrations after a process migration by exploiting computation dependencies. The objective was to minimize the NUMA effects on performance in migrating threads, memory pages, and both [8]. When a process gets migrated from one node to another, its memory may get scattered on many nodes due to load-balancing operations. To minimize contention for remote memory access, Chiang et al. improve the kernel's inter-node load balancing by migrating appropriate processes/ threads to remote nodes.

Since the inter-node page migration is a costly affair, they improved the inter-node load balancing mechanism of Linux to minimize inter-node memory access. This is made possible by selecting the processes for migration while keeping their memory usage in view, such that the selected processes use the minimal number of page frames and/or share the minimum amount of memory with other processes [2]. Many other researchers have also proposed and discussed the techniques for enhancing the efficiency of the linux load balancing technique and, consequently, the efficiency of NUMA systems. However, in light of the intricacies of load-balancing operations and ongoing advancements of NUMA machines, effective and efficient load-balancing techniques need to be developed to minimize the overheads of load-balancing. The work presented in this paper focuses on performing the process migration considering the originating nodes of the processes. It will make a significant contribution towards the performance improvement of NUMA Multiprocessor systems by developing cutting-edge load balancers.

4. Origin Aware Load Balancing

The novel load balancing algorithm proposed herein improves the performance of the existing load balancer by reducing the memory access overheads. While executing on a particular processor, the proposed load balancer prefers to pull the processes belonging to that processor/ node OR pushes the processes belonging to distant nodes (if present in the runqueue of the current processor) back to their parent processor/ node or a neighbour node. This approach reduces the memory access overheads to zero or minimum and, therefore, achieves significant performance gain. A load balancer, therefore, should try to avoid migration of processes to distant nodes in order to avoid indirect overheads of load balancing, viz., the remote memory access and cache-miss overheads (Liu, 2018).

The Process Selection and Process Placement policy incorporated in the Origin Aware Load Balancing Algorithm attempts to ensure the execution of any process on a processor in its parent node or nearby node for the maximum of its life span. The Origin Aware Load Balancer is composed of two components: the Receiver-Initiated Load Balancing (RI-LB) component and the Sender-Initiated Load Balancing (SI-LB) component. It performs the receiver-initiated load balancing and sender-initiated load balancing in alternate cycles of periodic load balancing. The proposed load balancer makes use of the following process selection and placement policy.

4.1. Process Selection & Placement Policy

The Receiver-initiated load balancing is performed similarly to linux load balancing but with a different process selection policy used. In RI-LB, according to the process selection policy, out of the processes eligible for migration from the busiest processor at a particular sched domain level, the processes that originated on the current processor/node are first migrated, followed by the other processes. In the Sender-initiated load balancing cycle, the load balancer attempts to send the processes back to their parent nodes while performing the load balancing. The load balancer checks the current runqueue to find whether it contains the processes originating on distant processors/nodes.

For each of such processes, if found, the possibility of sending it back to its parent processor /node is examined. If the current processor is more loaded than the parent processor, the selected process is migrated (pushed) to its parent processor; otherwise, it is migrated to another processor of the parent node, if possible. The operation is repeated for all processors of the current node and then for each remaining node. This policy balances the load as well as makes the memory access time of the migrated process, a minimum. As per the above load balancing methodology followed by the

proposed algorithm, a processor can offload its processes to their respective parent nodes, thereby reducing the memory access time of the migrated processes. However, sometimes, the parent processor or the node, due to being overloaded, is not able to accept the processes from the other processors. To address this possibility, two variants of the SI-LB component of the Origin Aware Algorithm have also been developed, as described in the following sub-sections.

4.2. Origin Aware Load Balancing (SI-LB): Variant1

When it is not possible to migrate a process to its parent processor/node, the parent processor or some other processor of the parent node is checked to find the presence of a process belonging to the current processor, and if found, the two processes are exchanged, i.e., the two processes are migrated to their respective parents.

4.3. Origin Aware Load Balancing (SI-LB): Variant2

If it is impossible to migrate a process of the current processor to its parent processor or node even after performing the load balancing through SI-LB variant1, loads of the nearest neighbour nodes of the parent node are examined to find the possibility of migration. If possible, the process will be migrated. Otherwise, this procedure is repeated for the next level of hierarchy (upto the sched_domain hierarchy, which is one or two levels below the level to which the current Node belongs relative to the parent Node of the process). In this way, the Origin Aware Load Balancing algorithm makes all possible attempts to minimize memory latencies of the processes by making them execute on their parent or neighbour nodes.

4.4. Functioning of the Origin Aware Load Balancer: An Illustration

To illustrate the functioning of the Sender-initiated load balancing component of the Origin Aware Load Balancer,we consider a load balancing scenario shown in Figure 6 and the load balancer executing on processor P0 belonging to Node N0. When the sender-initiated load balancing cycle begins, the load balancer executing on P0 finds few processes in its runqueue, originating on distant nodes, as shown in Table 3. As per the policy, the load balancer migrates the farthest originated processes, p45 and p118, to the respective parent processors (P10) in Node N5, the farthest node from the current Node N0. Next, while attempting to migrate p31, it was found that processor P11 was not underloaded, and P10 was not underloaded.

Thus, the two processes- p31 and p91are exchanged between P0 and P11 as per variant 1 of the algorithm. Next in sequence, an attempt is made to migrate p17 to P28 or P29, but this attempt fails; the exchange of processes between any of these processors and P0 could not be done, and therefore, as per variant2 of the algorithm, process p17 is migrated to a node (processor P24) nearer to its parent. Following this, process p1 is considered for migration; however, it could not be migrated, and therefore, it continues to execute on the current processor only. The remaining two processes, p9 and p37, are not migrated as p9 belongs to the next level node only; p37 belongs to the current processor only.

4.5. Origin Aware Load Balancing Algorithm

The Receiver-initiated load balancing component of the Origin Aware load balancer functions similarly to the linux load balancer, except that it prefers the processes belonging to the current processor or node for migration. The other component of the algorithm- the Sender-initiated load balancing component is described in this section.

The Origin Aware Load Balancer carries out the initial load balancing in the same way as done by linux. Regarding idle load balancing, when the load balancing operation is triggered by a processor having zero load, processes are pulled from heavily loaded processors, preferring the processes that originated on the current processor.

Table 3. Processes originated on other nodes and currently present in the runqueue of processor P0

Process	Node (and processor) on which the process was originally created
p37	N0 (P0)
p1	N3 (P6)
p9	N1 (P2)
p45	N5 (P10)
p31	N5 (P11)
p17	N14 (P28)
p118	N5 (P10)

Algorithm 1: Origin Aware Load Balancing (Sender-Initiated Load Balancing)

// Periodic Load Balancing: The following steps (steps 1 to 42) are carried out when the load balancer is invoked during alternate cycles of Periodic Load Balancing to perform Sender-initiated load balancing.

// For all the Nodes N=1 to n and processors P=1 to p of each Node, execute this code on each processor.

1. {
2. curr_node= N;
3. curr_processor= P;
4. For curr_processor, find all processes which are originated on other nodes (off-node processes) and presently exist in its runqueue;
5. x= no. of all such processes;

```
6.  if (x > 0)
7.  {
8.  For  processes K=1 to x  do   //for all off-node processes in the runqueue   of  curr_processor
9.  {
10. curr_process= process K;
11. target node=N' ;                        // N' is parent node of curr_process
12. curr_level_of_sched_domain= L      //Relative to N', curr_node is at L^th
    level of  sched_domain hierarchy
13. target_processor=P' ;                   //parent processor of curr_process
14. check the load of target_processor;
15. if (load of curr_processor > load of target_processor)
16. {
17. obtain lock on target_processor;
18. obtain lock on curr_processor;
19. migrate process K from curr_processor to target_processor;       //PUSH  migration
20. migrated=TRUE;
21. release lock on curr_processor;
22. release lock on target_processor;
23. }
24. else     //if curr_process can not be migrated to parent processor
25. { //Attempt to migrate the curr_process to some other processor, other than the parent processor, of the parent node
    (which is N')
26. target_node=N' ;
27. For all processors of target_node do
28. {
29. target_processor = next processor of target_node;
30. execute steps 14 to 23;
31. if (migrated)          //if curr_process is migrated to some  processor of  parent Node
32. go to step 39;
33. }
34. }    //End of If statement at step no. 15 (and the Else part at step no. 24)
35. if (not migrated)    //if curr_process could not be migrated to its parent Node
36. attempt migration of curr_process to parent Node by exchanging this process with some
    process  on the parent node, as per Variant1
37. if (not migrated)    //still if curr_process could not be migrated to its parent Node
38. attempt migration of curr_process to nearest possible neighbor of its parent node, as per Variant2
39. K= K+1;        //take up the next process
40. }              // End of FOR loop at step no. 8
41. }      // End of If statement at step no. 6
42. }      //End of Algorithm   (Sender-initiated component of the algorithm)
```

5. Simulation and Results

To assess the performance of the Origin Aware Load Balancer, experimentation was carried out using a NUMA Multiprocessor/Multicore system simulator for linux [19] for three NUMA Systems M1 to M3**,** with number of Nodes (N), number of processors per Node (P/N) and number of Memory Access Levels (MAL) as following:

- M1: N=16, P/N=2, MAL=6
- M2: N=16, P/N=4, MAL=6
- M3: N=8, P/N=2, MAL=3

For each system, different workloads were generated with combinations of,

- Number of processes,
- Process Execution time: Fixed average time or varying time
- Process arrival: same time, periodic or random
- Process type: CPU bound or IO bound or a mix of CPU and IO bound.

5.1. Turn Around Time and Performance Gain

The experimental results in terms of Turn Around Time (in ms) and Performance Gain (improvement in TAT in %) are presented in Tables 4 through 6 and also depicted in the graphs given after the corresponding Tables; W1, W2, W3 characterize the workloads.

5.2. Information Obtained from Traces of the Processes

Information obtained from the traces of processes regarding a number of processes migrated and executed on the nodes distant from respective parent nodes is presented in Table 7.

Traces of a few processes are also given in Table 8, depicting how processes were migrated back to parent Node or neighbour Node in Origin Aware Load Balancing.

5.3. Observations and Discussion

The experimentation results clearly show that the Origin Aware Load Balancer outperforms the Linux Load Balancer. For NUMA systems with varying configurations and different workload characteristics, it demonstrated improved average Turn Around Time ranging from 7 to 23 %.

The acquired performance gain can be attributed primarily to the minimized memory access overheads.

A few points of observation are as follows:

- For NUMA systems having a large number of MALs, the performance gain is higher for the obvious reason: in such systems, the probability of processes being migrated to far nodes is higher, and thus, more improvement is achieved in Origin Aware Algorithm.
- As compared to linux load balancing, less number of processes were migrated to far nodes; also, many of such processes were migrated back to their parent nodes, as shown in Table 8, resulting in improved TAT of the processes, very significantly.
- For I/O bound processes performance gain is relatively less as compared to CPU bound processes because of relatively less number of memory accesses done by the I/O bound processes.
- For workloads having a very small number of processes, the performance gain is either –ve or very insignificant. This is due to the algorithm's overheads for smaller workloads. Therefore, for very small workloads, variants of the proposed algorithm should not be invoked.

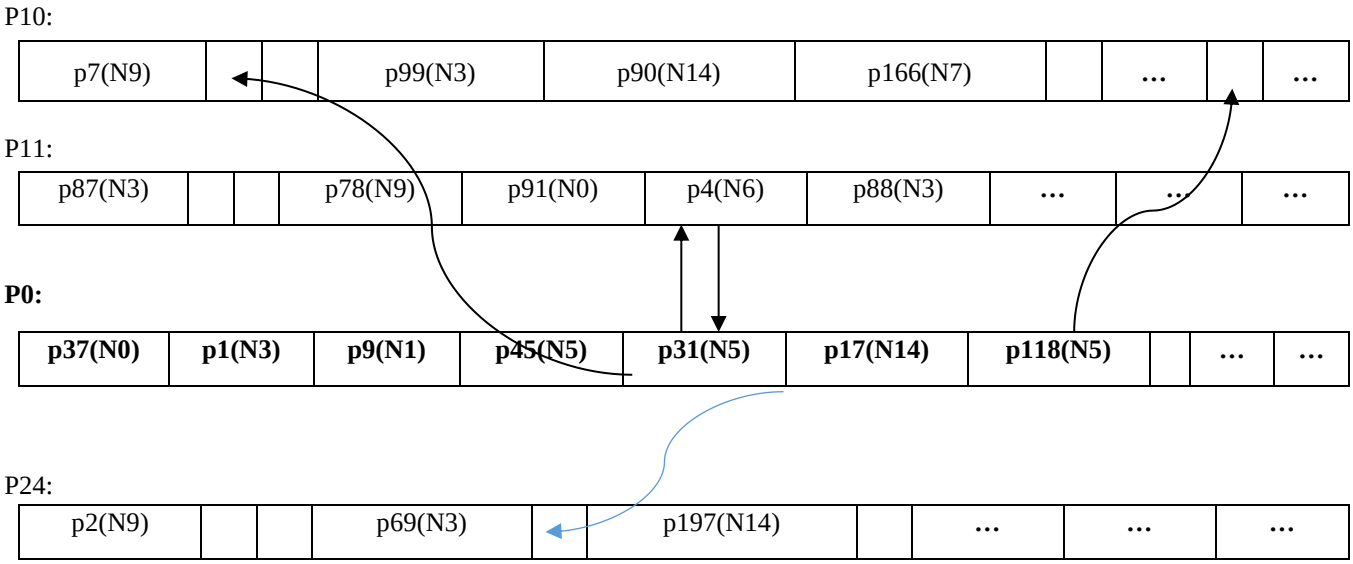

Fig. 6 Depiction of process migration in origin aware load balancing
(P0, P10…. are runqueues of Processors P0, P10 … and pi(Nj) is the process with pid=i & parent Node Nj)

Table 4. Turn Around Time of processes and performance gain for origin aware load balancing vs linux load balancing for NUMA system M1

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		
	TAT: Linux Algo.	Origin Aware Algo.	Perf. Gain (%)	TAT: Linux Algo.	Origin Aware Algo.	Perf. Gain (%)	TAT: Linux Algo.	Origin Aware Algo.	Perf. Gain (%)
25	241	253	-4.98	322	338	-4.97	454	477	-5.07
50	367	359	2.18	409	387	5.38	684	601	12.13
100	613	573	6.53	610	491	19.51	1019	821	19.43
200	1257	1088	13.45	989	791	20.02	1720	1391	19.13
300	1834	1520	17.13	1399	1121	19.87	2451	1936	21.01
400	2380	2033	14.58	1804	1478	18.07	3142	2437	22.44
500	2751	2355	14.39	2199	1787	18.74	3847	2977	22.62

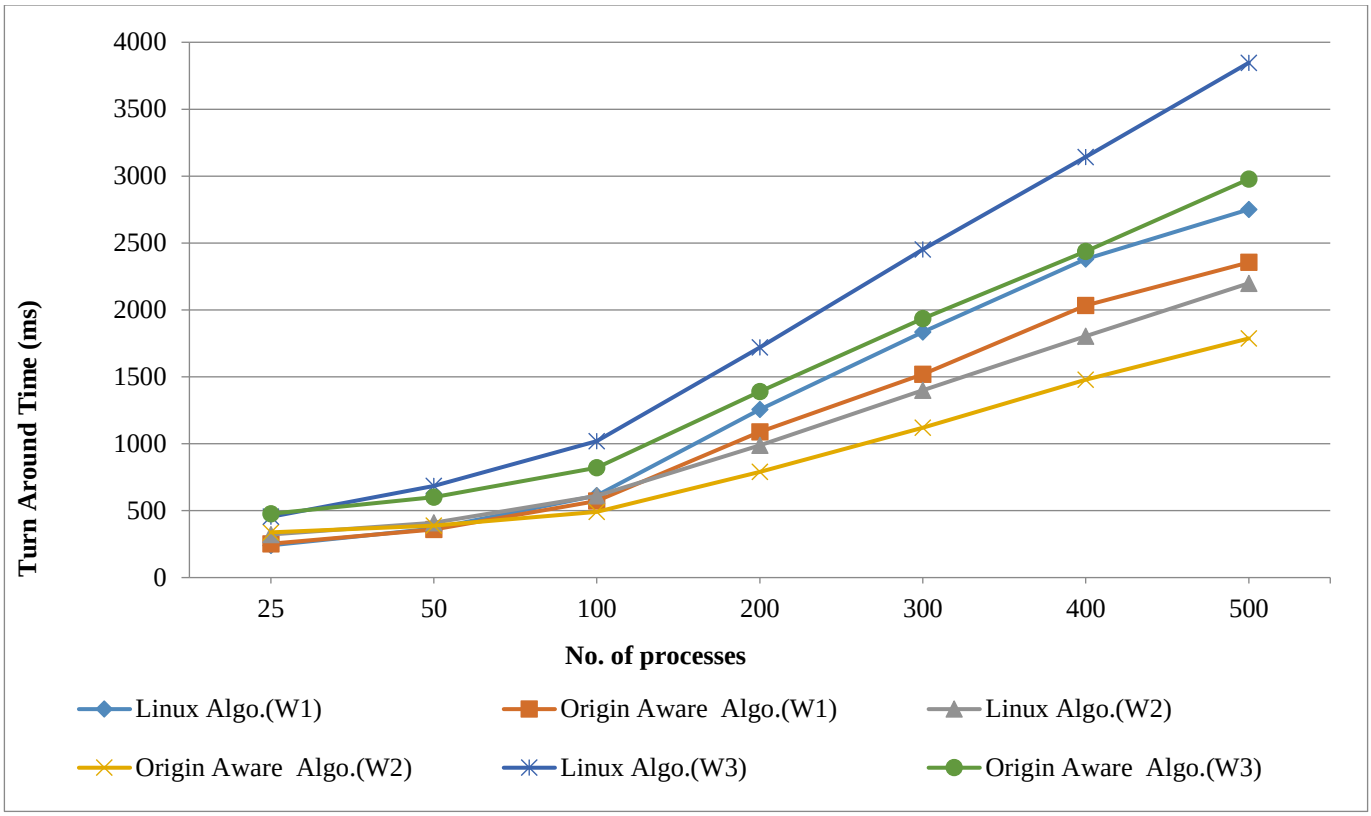

Fig. 7 Turn Around Time of processes for origin aware load balancing vs Linux load balancing for NUMA system M1

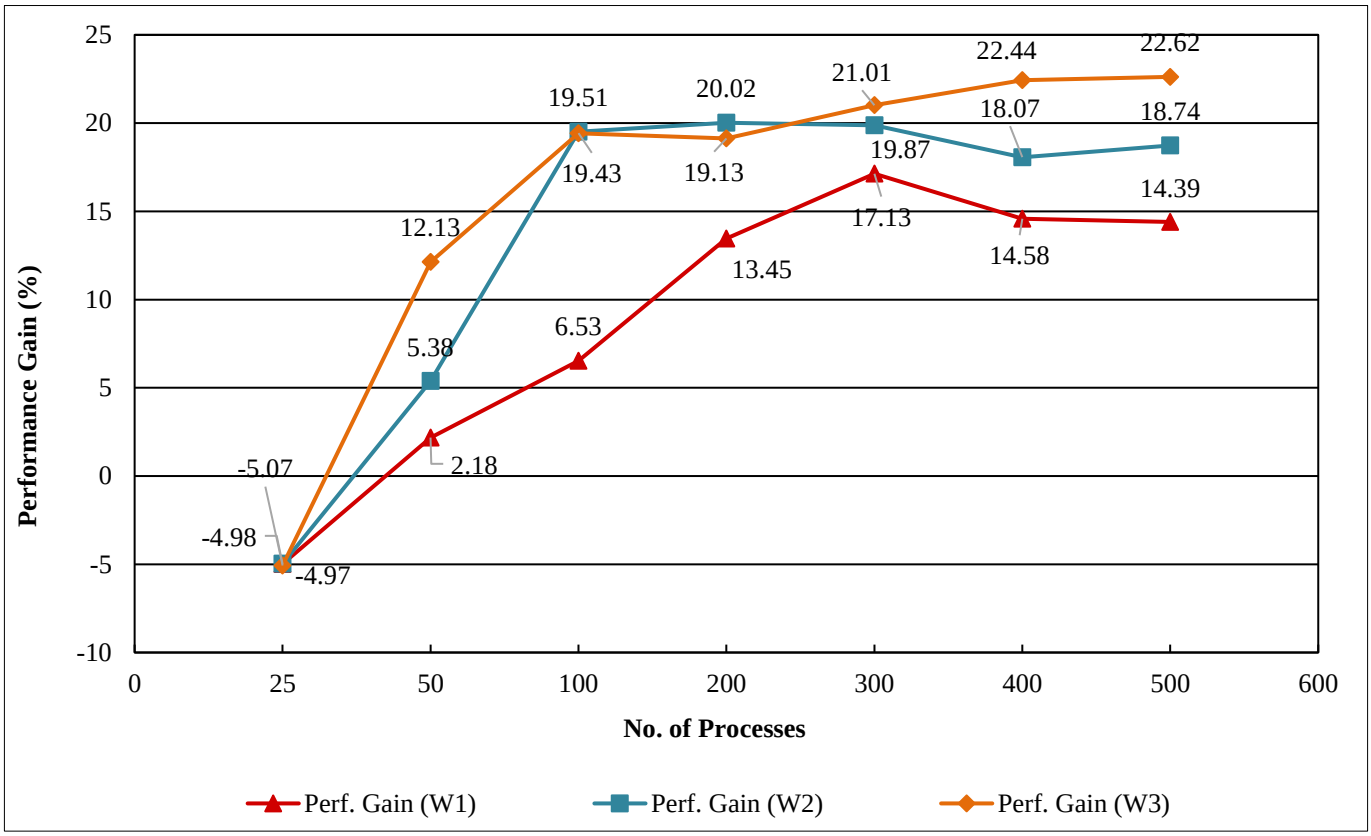

Fig. 8 Performance gain for origin aware load balancing over linux load balancing for NUMA system M1

Table 5. Turn Around Time of processes and performance gain for origin aware load balancing vs Linux load balancing for NUMA system M2

No. of processes	W1- Process type: CPU Bound; Execu time: 200 ms; Arrival- Random			W2- Process Type: Mix of CPU & IO Bound; Execu Time: Varying (50-400 ms); Arrival- random		
	TAT: Linux Algo.	TAT: Origin Aware Algo.	Perf. Gain (%)	TAT: Linux Algo.	TAT: Origin Aware Algo.	Perf. Gain (%)
25	441	442	-0.22	568	590	-3.87
50	515	517	-0.38	557	501	10.05
100	670	592	11.64	678	565	16.67
200	861	738	14.29	997	785	21.26
300	1085	889	18.06	1253	1003	19.95
400	1399	1123	19.73	1538	1238	19.51

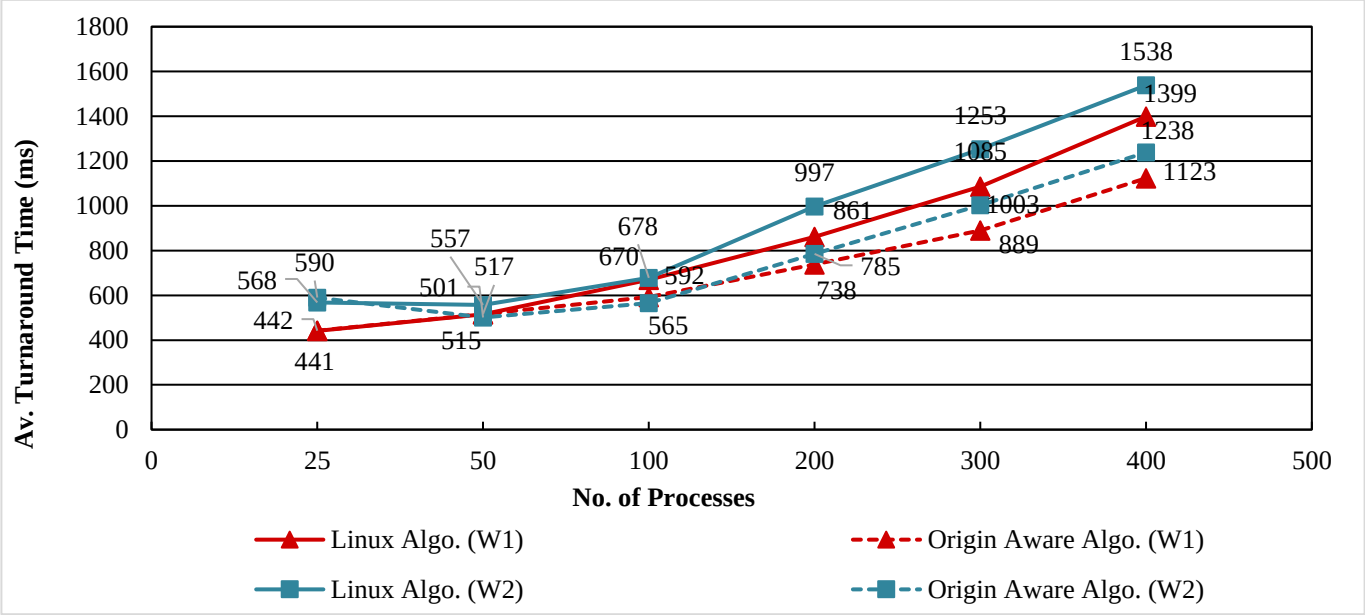

Fig. 9 Turn Around Time of processes for origin aware load balancing vs Linux load balancing for NUMA system M2

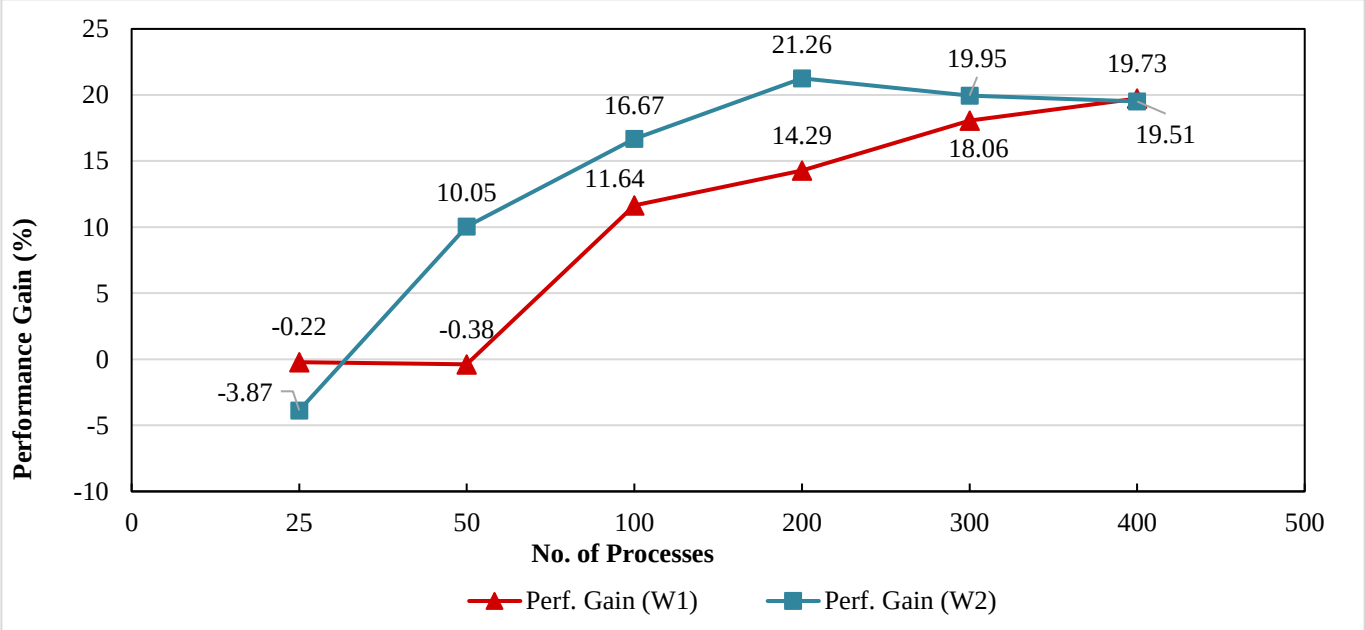

Fig. 10 Performance gain for origin aware load balancing over linux load balancing for NUMA system M2

Table 6. Turn around time of processes and performance gain for origin aware load balancing vs Linux load balancing for NUMA system M3

No. of processes	W1- Process type: CPU bound; Execu time: 200 ms; Arrival- random			W2- Process type: CPU bound; Execu time: 300 ms; Arrival- random			W3- Process type: IO bound; Execu time: 200 ms; Arrival- random		
	TAT: Linux Algo.	TAT: Origin Aware Algo.	**Perf. Gain (%)**	TAT: Linux Algo.	TAT: Origin Aware Algo.	**Perf. Gain (%)**	TAT: Linux Algo.	TAT: Origin Aware Algo.	**Perf. Gain (%)**
100	785	734	6.50	1430	1402	1.96	533	495	7.13
200	1498	1324	11.62	2764	2558	7.45	978	849	13.19
300	2240	1930	13.84	4058	3711	8.55	1320	1215	7.95
400	2880	2525	12.33	5441	4843	10.99	1726	1558	9.73

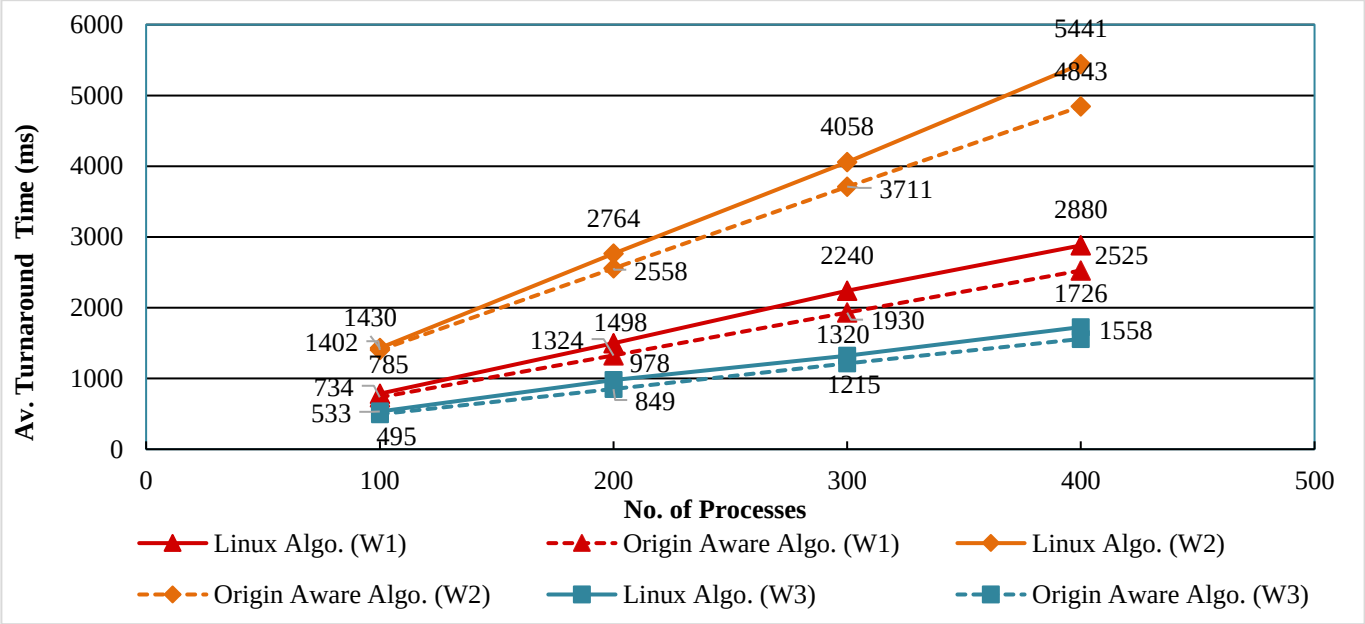

Fig. 11 Turn around time of processes for origin aware load balancing vs Linux load balancing for NUMA system M3

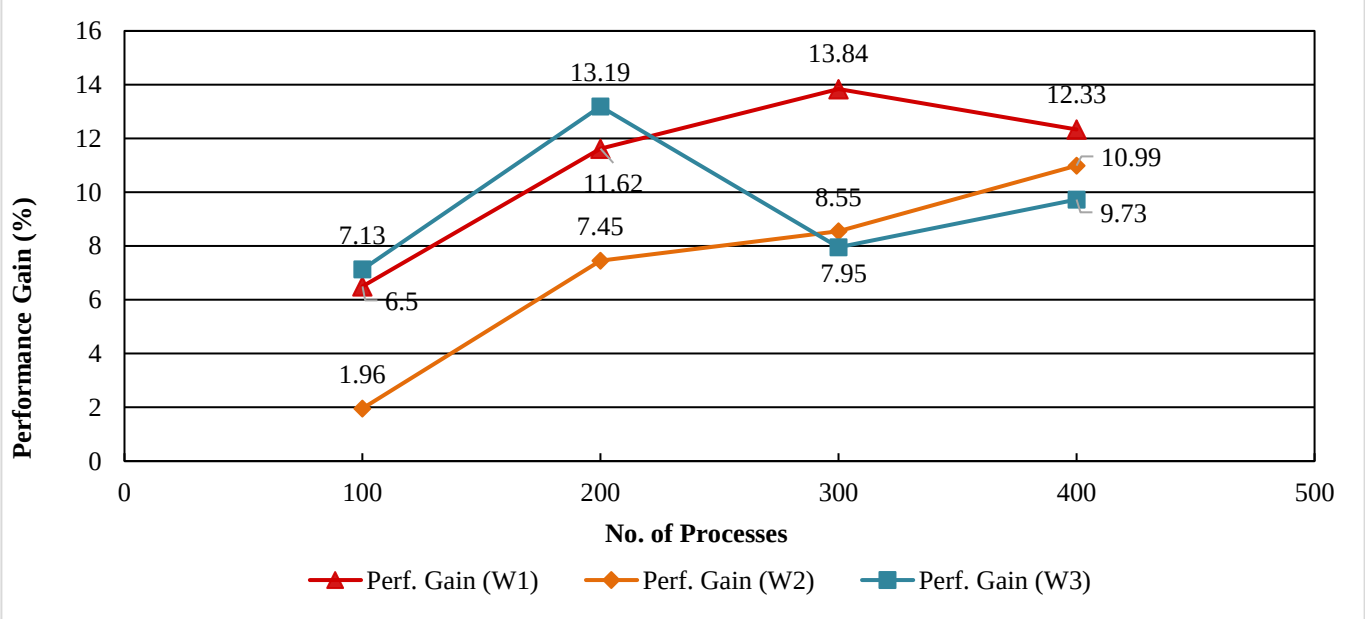

Fig. 12 Performance gain for origin aware load balancing over linux load balancing for NUMA system M3

Table 7. No. of processes executed on the processors distant from their respective parent nodes in origin aware load balancing as compared to linux load balancing

Total No. of Processes	Processes that Remained Far from Parent Nodes (in Linux Load Balancing)		Processes that Remained Far from Parent Nodes (in Origin Aware Load Balancing)	
	No. of Such Processes	% of Such Processes	No. of Such Processes	% of Such Processes
100	46	46 %	22 (of which 09 were at II level Node)	22 %
200	96	48 %	49 (of which 20 were at II level Node)	24.5 %
300	129	43 %	53 (of which 19 were at II level Node)	17.7 %

Table 8. Traces of few processes (obtained for a workload of 300 processes) showing the migration pattern in origin aware load balancing

Process-id	Node (and Processor) on Which the Process was Originally Created	Migration Took Place from Which Processor to Which Processor (and the Node on Which the Process Executed Most of the Time)	Process Migrated Back to Which Node, from The Far Node
87	N1 (P2)	P2-P17; P17-P24 (N12)	II level Node (Nearest Neighbour Node)
184	N14 (P29)	P29-P28; P28-P26; P26-P29 (N14)	Parent Node
251	N12 (P24)	P24-P12; P12-P24 (N12)	Parent Node
183	N1 (P2)	P2-P13; P13-P25 (N12)	II level Node (Nearest Neighbour Node)
197	N7 (P15)	P15-P19; P19-P14; P14-P15 (N7)	Parent Node
240	N5 (P11)	P11-P25; P25-P11(N5)	Parent Node
241	N2 (P5)	P5-P30; P30-P5 (N2)	Parent Node
246	N6 (P12)	P12-P9; P9-P13 (N6)	Parent Node
281	N15 (P30)	P30-P31; P31-P19; P19-P4 (N2)	II level Node (Nearest Neighbour Node)
174	N7 (P15)	P15-P14; P14-P8;P8-P14 (N7)	Parent Node

6. Conclusion

In this paper, we investigated the process migration mechanism of the linux load balancer and proposed an Origin Aware Load Balancing Algorithm for NUMA Multiprocessor Systems based on the modified process placement policy. It ensures that the processes are either not migrated too far from their originating nodes or are brought back to those nodes, if they are migrated at all. The proposed algorithm greatly decreases memory access overheads and thereby improves performance significantly.

References

[1] Martin J. Bligh et al., "Linux on NUMA Systems," *Linux Symposium*, vol. 1, pp. 89-102, 2004. [Google Scholar] [Publisher Link]

[2] Mei-Ling Chiang et al., "Enhancing Inter-Node Process Migration for Load Balancing on Linux-Based NUMA Multicore Systems," *2018 IEEE 42nd Annual Computer Software and Applications Conference*, Tokyo, Japan, pp. 394-399, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[3] M. Correa et al., "*Multilevel Load Balancing in NUMA Computers*," Technical Report Series, PPGCC-FACIN-PUCRS, Brazil, no. 49, pp.1-22, 2005. [Google Scholar] [Publisher Link]

[4] Alexey Paznikov, "Optimization of Thread Affinity and Memory Affinity for Remote Core Locking Synchronization in Multithreaded Programs for Multicore Computer Systems," *Vibroengineering Procedia*, vol. 12, pp. 213-218, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[5] Matthew Dobson et al., "Linux Support for NUMA Hardware," *Linux Symposium*, pp. 169-184, 2003. [Google Scholar] [Publisher Link]

[6] Christoph Lameter, "Local and Remote Memory: Memory in a Linux/NUMA System," *Linux Symposium*, pp. 1-25, 2006. [Google Scholar] [Publisher Link]

[7] Ilaria Di Gennaro, Alessandro Pellegrini, and Francesco Quaglia, "OS-Based NUMA Optimization: Tackling the Case of Truly Multithread Applications with Non-Partitioned Virtual Page Accesses," *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Cartagena, Colombia, pp. 291-300, 2016. [CrossRef] [Google Scholar] [Publisher Link]

[8] Isaac Sánchez Barrera et al., "Reducing Data Movement on Large Shared Memory Systems by Exploiting Computation Dependencies," *Proceedings of the 2018 International Conference on Supercomputing*, Beijing, China, pp. 207-217, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[9] Ye Liu, Shinpei Kato, and Masato Edahiro, "Optimization of the Load Balancing Policy for Tiled Many-Core Processors," *IEEE Access*, vol. 7, pp. 10176-10188, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[10] Matthias Diener et al., "Kernel-Based Thread and Data Mapping for Improved Memory Affinity," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 9, pp. 2653-2666, 2016. [CrossRef] [Google Scholar] [Publisher Link]

[11] Jean-Pierre Lozi et al., "The Linux Scheduler: A Decade of Wasted Cores," *Proceedings of the Eleventh European Conference on Computer Systems*, London United Kingdom, pp. 1-16, 2016. [CrossRef] [Google Scholar] [Publisher Link]

[12] Omar Kermia, and Yves Sorel, "Load Balancing and Efficient Memory Usage for Homogeneous Distributed Real-Time Embedded Systems," *2008 International Conference on Parallel Processing – Workshops*, Portland, OR, USA, pp. 39-46, 2008. [CrossRef] [Google Scholar] [Publisher Link]

[13] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova, "Thread and Memory Placement on NUMA Systems: Asymmetry Matters," *2015 USENIX Annual Technical Conference (USENIC ATC '15)*, Santa Clara, CA, USA, pp. 276-289, 2015. [Publisher Link]

[14] Laércio L. Pilla et al., "A Hierarchical Approach for Load Balancing on Parallel Multi-Core Systems," *2012 41st International Conference on Parallel Processing*, Pittsburgh, PA, USA pp. 118-127, 2012. [CrossRef] [Google Scholar] [Publisher Link]

[15] 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*, vol. 12, no. 4, pp. 1-24, 2015. [CrossRef] [Google Scholar] [Publisher Link]

[16] Suresh B. Siddha, Sched: New Sched Domain for Representing Multicore, 2006. [Online]. Available: https://lwn.net/Articles/169277/

[17] Li Wang et al., "NUMA-Aware Scalable and Efficient In-Memory Aggregation on Large Domains," *IEEE-Transactions on Knowledge and Data Engineering*, vol. 27, no. 4, pp. 1071-1084, 2015. [CrossRef] [Google Scholar] [Publisher Link]

[18] Saleh A. Khawatreh, "An Efficient Algorithm for Load Balancing in Multiprocessor Systems," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 3, pp. 160-164, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[19] Mei-Ling Chiang et al., "Memory-Aware Kernel Mechanism and Policies for Improving Inter-Node Load Balancing on NUMA Systems," *Software: Practice and Experience*, vol. 49, no. 10, pp. 1485-1508, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[20] Mei-Ling Chiang, and Wei-Lun Su, "Thread-Aware Mechanism to Enhance Inter-Node Load Balancing for Multithreaded Applications on NUMA Systems," *Applied Sciences*, vol. 11, no. 14, pp. 1-22, 2021. [CrossRef] [Google Scholar] [Publisher Link]