Capability Aware Dynamic Load Balancer for Asymmetric NUMA Multicore Processors

D.A. Mehta
Shri G S Institute of Technology and Science
Indore, MP

Priyesh Kanungo, PhD
School of Computer Science, Devi Ahilya
Vishvavidyalaya, Indore, MP

ABSTRACT
The different cores of state-of-the artmulticore processors are performance asymmetric: they are different in terms of their clock speeds and other capabilities. Such heterogeneous multicore processors pose challenges to existing Dynamic Load Balancers in achievingoptimum performance.A load balancer taking the load balancing related decisions assuming all cores to be homogeneous, introduces unnecessary overheads of time, does not exploit the capabilities of higher performance cores, and consequently fails to achieve the possible performance improvement.We, therefore, propose a Capability Aware Dynamic Load Balancer which performs efficient load balancing for asymmetric multicore processors by addressing the aforesaid issues. Considering the difference in clock speeds as the heterogeneity among different cores, the proposed load balancer improves the Turn Around Time of processes significantly as compared to Asymmetry unaware linux load balancer. The results of experimentation exhibit the performance gain in the range of 4-9%, for three different multicore systems having 32, 64 and 128 cores respectively.

Keywords
Dynamic Load Balancing, DLB, Load Balancer, NUMA, S*peed* Core

1. 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.

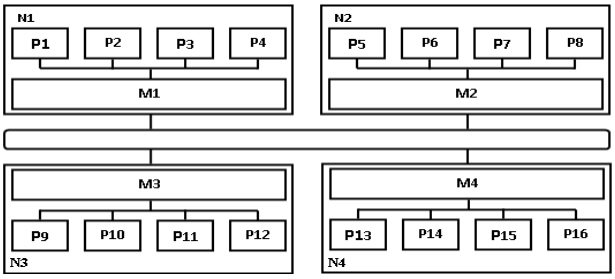

Figure 1: NUMA System with 4 Nodes, 16 Processors and 2 Memory Access Levels[N1, N2 … are Nodes; P1, P2 … are Processors; M1, M2 … are Memories]

It is said to have 2 Memory Access Levels (MALs) due to twodifferent memory latencies: (i) When a processor accesses memory in its own node (ii) When the processor accesses any memory outside its node [5].

1.1 Dynamic Load Balancing
Linux, a widely used operating system for NUMA systems, implements separate runqueues 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 sched domain hierarchy for the system shown in Figure 1. The lowest level sched domains are called CPU/Coredomains. 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. Thus for NUMA system with two memory access levels, there will be two levels in the sched domain hierarchy and the node domain will comprise of all the nodes of the system, as shown in Figure 2 [5].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 lowest level domain, scheduling groups will be all the processors in its node; and at higher levels, scheduling groups will be all the nodes at that level.

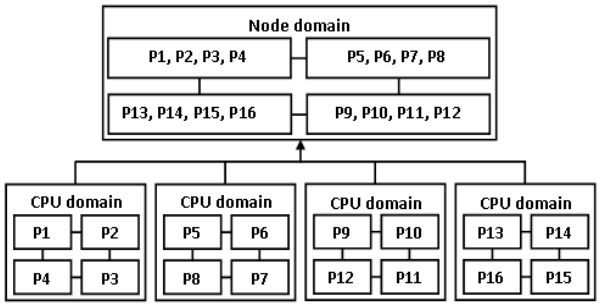

Figure 2: Sched Domain Hierarchy for NUMA System with Two Memory AccessLevels

Load balancer, which runs on each processorseparately, is invoked in three different situations and performs the load balancing as explained below [15] [17]:
(i) 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 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) (ii) When a task is newly created or woke-up through system calls fork(), exec(), clone(), wakeup() : In this condition, the task is moved to the least loaded processor of the least loaded scheduling group (node) in its current domain (iii) When a processor becomes idle: In this condition, *idle load balancing* is performed by the idle processor starting from the parent domain and moving upward in the sched domain hierarchy; it selects the most loaded scheduling group (node) in the current domain and migrates tasks from the most loaded processor to this processor.

It is obvious from the foregoing description that large overheadsof time are involved in performing the dynamic load balancing. Though these overheads are inevitable and not avoidable always, an efficient load balancer should attempt to minimize them by finding the conditions under which unnecessary attempts of load balancing and process migrations may be avoided. However, an asymmetry unaware load balancer, instead of minimizing the overheads, may aggravate the problem of load balancing overheadsas it may not exploit the capabilities of faster cores (*speed* cores) in performing load balancing activities (as detailed in Section 2), and therefore, will not achieve optimum performance in terms of cores' utilization and Turn Around Time (TAT) of the processes.

The objective of our work, therefore, is the design of a Capability Aware Load Balancer which performs the load balancing in accordance with the capabilities of different cores (in our case, faster/slower clock speeds) to achieve the better performance by avoiding frequent load balancing attempts and taking proper load balancing decisions.

2. PITFALLS IN ASYMMETRY UNAWARE LOAD BALANCING ALGORITHM AND SCOPE OF ITS IMPROVEMENT

In this sectionthe reasons of more overheads incurred in an asymmetry unaware load balancer are analysed first, and then the scope of its improvement is explored. During periodic or *idle load balancing*, time overheads are incurred in finding the load of various processors/nodes, comparing it with the load of the processor performing the load balancing, and migration of the processes, if needed.A load balancer unaware of the presence of asymmetric cores in a multicore processor will cause too frequent load balancing attempts done by faster cores due to the fact that these cores will speedily complete the execution of the tasks allocated to them and will create load imbalance across the various cores frequently. In addition, this load balancer will also not take certain load balancing decisions in accordance with the capabilities of the cores, for example, allocation of similar cores to different threads of the same process, allocation of faster cores to CPU bound tasks etc. As a consequence of all these inappropriate decisions, there will be more overheads resulting into non-optimum or even degraded performance.

In order to further understand the aforesaid issues, we performed the load balancing (through simulation) on various NUMA multicore systems having asymmetric cores (half of the cores, faster cores and half of the cores, slower or normal cores) using the asymmetry unaware linux load balancer. A careful analysis of the process traces and simulation results revealed that due to non-exploitation of the faster cores available, the asymmetry unaware load balancer incurs more load balancing overheads, majorly the *idle load balancing* overheads and, therefore, is not able to achieve the optimum performance. The findings are analysed and explained through an example in which the load balancer executing on a normal core C7 attempts load balancing against a *speed* core C0. Figure 3 depicts the scenario of load balancing. For simplicity it is assumed that *load* of each process is same, in this example.

As shown in the Figure, during a load balancing cycle, the load balancer finds a *speed* core C0 to be overloaded as compared to it: C7 at time t=t0 hasa load of 2 processes whereas C0 has a load of 10 processes. The load balancer therefore pulls 4 processes- p6, p7, p8 and p9, to equalize the load between the two cores. After certain amount of time, say at time t=t1, C7 still has these processes partially executed, in its runqueue , alongwith two other processes, however, CO being the *speed* core finishes the execution of all the processes in its runqueue in the time period t0-t1. Since the runqueue of C0 becomes empty, it initiates an *idle load balancing* and incidentally pulls the processes p7, p8 and p9 fromC7: the same processes which were pulled by C7 from C0 during the current load balancing cycle. Same situation occurs at t=t2 when the runqueue of C0 again becomes empty and it pulls one process- p4, from C7. In this manner the *speed* core C0 perform *idle load balancing* many times before the occurrence of next regular load balancing cycle. Moreover, it pulls the CPU bound processes p7 and p9 from the faster core CO even when IO bound processes p0 and p99 were present.

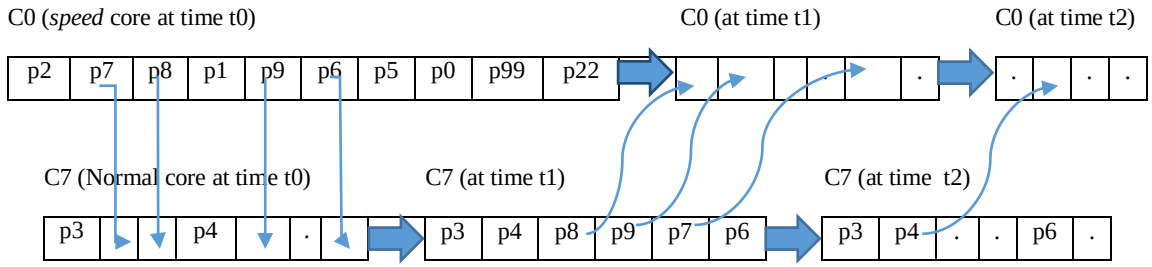

Figure 3: Load Balancing Scenario during a load balancing cycle, for Asymmetry Unaware Load Balancer (C0, C7 are Runqueues of Cores C0, C7 ; pi is process with pid=i)

From the aforesaid observations about the functioning of the load balancer, it can be noted that: (i) More no. of *idle load balancing* attempts are done by the *speed* core during a single load balancing cycle, resulting into increased overheads(ii) Memory access time may increase because the pages of some of the processes pulled, may be lying on the Nodesfrom which they are migrated (iii) Cache-miss overheads may increase, because the cache of the migrated processes, located on their

previous cores, will become invalid(iv) TAT of CPU bound processes is likely to be more because they are not allowed to fully execute on the faster core C0.
The same situation as described above will arise when a *speed* core will perform load balancing against a normal core. To avoid the overheads arising due to unawareness of the load balancer about the difference in clockspeeds of the cores, the load balancer executing on normal core should have pulled lessno. of processes (processes with less load) from the *speed* core and that too the IO bound processes, and vice-versa. If this was done, unnecessary overheads of frequent *idle load balancing* and cache miss & memory latency would not have come in picture.
Based on these findings and threads obtained for improvement, an efficient load balancer for asymmetric multicore processors has been proposed in this paper.

3. RELATED WORK

Asymmetric multicore processors pose unique challenges to existing load balancers which traditionally assume all the cores to be homogeneous, and therefore do not work well for asymmetric architectures. Many researchers have identified the relevant issues and addressed them in their work. Significant contributions of some of the researchers are presented below:
Li et al. proposed an Asymmetric Multiprocessor Scheduler (AMPS) which was composed of asymmetry-aware load balancing, faster-core-first scheduling, and NUMA-aware migration, for performance-asymmetric multi-core architectures. Their asymmetry-aware load balancing took advantage of distinct computing powers of the heterogeneous system; Faster-core-first scheduling policy of their scheduler enabled threads to run on more powerful cores whenever they are under-utilized [13].An algorithm called speed balancing was proposed by Hofmeyr et al. for asymmetric multicore processors. Instead of the use of weights as done by the linux load balancer, they substituted the computation for load balancing to be based on speed, where speed is defined as the CPU elapsed time divided by the actual wall clock time. Instead of balancing runqueue length, their load balancer balanced the time a thread had executed on faster and slower cores. The authors argued that the migration to a faster core will be able to effectively compensate for the migration overheads across caches[8].
Kim et al. in [11] emphasize the need tomatch each application with the best core type in asymmetric multi-core systems. Based on their study they state that uneven core capability is inherently unfair to threads and causes performance variance, as applications running on fast cores receive higher capability than applications on slow cores. Scheduling policies were, therefore, proposed by them which guarantee a minimum performance bound while improving the overall throughput and reducing performance variation. Although the work of the authors was towards scheduling policies for asymmetric multicore processors, it is obvious that the load balancing should also be performed taking the advantages of faster cores.
In [15], authors propose an optimized load balancing policy for multi-threaded applications executing on recently designed Tiled multicore processors, in which processing cores are fitted onto a single chip and are interconnected via mesh-based networks. A load balancer designed for traditional multicore systems and unaware of this new architecture, might introduce the penalty of cache misses because of the more threads sharing the same tile (processing core), and the contention for memory controllers due to cache misses. Thus, authors implemented an optimized load balancing policy for tiled many-core processors- KNL and the TILE-Gx72. This work is although not directly related to asymmetric multicore processors, but certainly states the need and advantage of an architecture aware load balancer.
Many other researchers have done similar work. The commonality in their work is the use of an appropriat core for a particular purpose, depending on its capability so as to optimize the system performance. The work presented in our paper will also be a contribution towards the same objective.

4. PROPOSED CAPABILITY AWARE LOAD BALANCING ALGORITHM

The proposed Capability Aware Load Balancer basically performs all the load balancing activities keeping in view the speed of thedifferent cores and taking the advantage of available faster coresof the multicore processors. The load balancer will be invoked in the same conditions under which linux load balancer is invoked, however, it will function differently in certain aspects as per the following load balancing policies:

4.1 Load Balancing Policies Incorporated in theCapability Aware Load Balancer

4.1.1 Process Migration Policy:

The proposed load balancer performs the periodic load balancing as usual, however while performing the load balancing between two dissimilar cores, it pulls more processes (having double the load as compared to current core, assuming the *speed* core to be two times faster). Further, it prefers to pull the CPU bound processes, if it is executing on a *speed* core; and vice-versa. This policy will avoid the unnecessary *idle load balancing* attempts as were incurred in an asymmetry unaware algorithm. Figure 4 illustrates the same, for the example taken in Section 2.
It can be observed that no*idle load balancing* attempts take place during the load balancing cycle, and IO bound processes are preferred for migration by the normal core, allowing the CPU bound processes to execute on the faster core.

4.1.2 Task Selection Policy:

(i) While selecting the tasks for migration on *speed* coresfrom the normal cores, priority is given to CPUbound tasks to take advantage of the speed (ii) In a multithreaded application environment, all threads of a particular process are kept on similar cores (*speed* cores, if possible) so that the process completion is not held-up due to slower threads.

4.1.3 No Load Balancing/No Operation Policy:

When task to *speed*core ratio is <= 1, i.e., when no. of tasks in the system are less than or equal to no. of *speed*cores, the load balancer pulls all the tasks from the slower core, if it is executing on *speed* core;and pushes all the tasks to *speed* core, if it is executing on the slower core. Moreover, it does not execute on idle slower cores till the task/*speed*core ratiodoes not become more than 1.

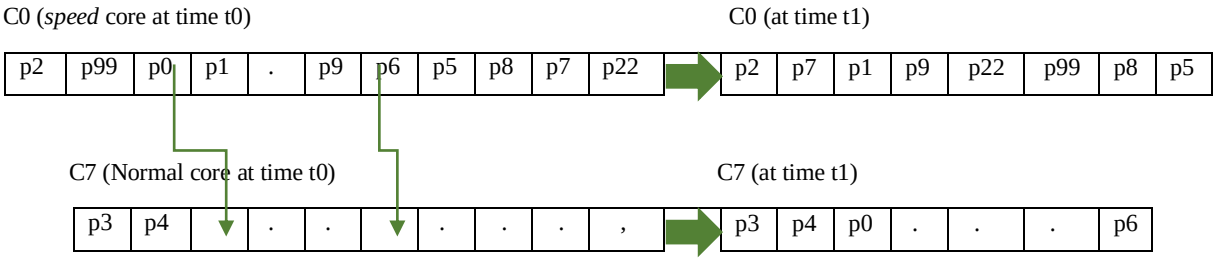

Figure 4: Load Balancing Scenario during a load balancing cycle, for Capability Aware Load Balancer
(C0, C7 are Runqueues of Cores C0, C7 ; pi is process with pid=i)

4.1.4 Selection of Core for Load Balancing:

If faster cores are free, they are employed for performing the load balancing.

The formal description of the proposed load balancing algorithmis as following:

Algorithm 1 : Capability Aware Load Balancing

For all Nodes of the system N=1 to n and all cores C=1 to c of each Node, carry out the following steps:

```
1. {
2.   if ((systemwide task to speed core ratio is <=1)
         and (current core is a normal core)) then
3.           no load balancing to be done by the current core;
             // this normal core eventually become idle core
               and hence it is not required to perform load
               balancing till task to speed core ratio is <=1
4.   for MAL=1 to max_memory_access_levels do
5.   {
6.     if (MAL==1) then
7.     {
8.       core_performing_LB = CC;
           //CC is the idle core or the first core
9.       find the load of all cores of curr_node, except the
         core_performing_LB;
10.      find the busiest core;    // core having highest load
           // while finding the busiest core, load of a speed
              core is considered as half of its actual load
11.    }
12.    else   //if MAL is > 1
13.    {
14.      core_performing_LB = CC;
           //CC is the idle core or the first core
15.      find the busiest scheduling group out of all the
         scheduling groups (all nodes) at memory access
         level MAL;
           // while calculating the load of any node, load of a
              speed core is considered half of its current load
16.      find the busiest core of the busiest node
         (scheduling group with highest load);
17.    }                  //end of if statement at step no. 6
18.    if (core_performing_LB is speed core .and.
           busiest core is NOT the speed core) then
19.    {
20.      LB_core_load = load of core_performing_LB/2;

21.      target_core_load = load of busiest core;
22.      if (systemwide task to speed core ratio is <=1) then
23.      {
24.        migrate all processes from busiest core to
           core_performing_LB;
25.        go to step 49;          //go to next MAL
26.      }
27.  }
28.  else
29.    if (core_performing_LB is slower core .and. busiest
       core is speed core) then
30.    {
31.      target_core_load = load of busiest core/2;
32.      LB_core_load = load of core_performing_LB;
33.      if (systemwide task to speed core ratio is <=1) then
34.      {
35.        migrate all processes from core_performing_LB
           to busiest core;
               // perform a PUSH migration to speed core
36.        go to step 49;          //go to next MAL
37.      }
38.  }
39.  endif;      //end of if statement at step no. 18
40.  if (LB_core_load  <  target_core_load) then
41.  {
         // compare the load of the core_performing_LB
            with that of the busiest core
42.    obtain lock on target_core;
         // busiest core is the target core
43.    obtain lock on core_performing_LB;
44.    select appropriate no. of processes for
       migration (in accordance with the speed of the
       core_performing_LB), preferring CPU-bound
       processes if core_performing_LB is speed core;
45.    migrate the selected processes from busiest core to
       core_performing_LB;
         // pull the processes/threads from the busiest core
            till the load of the two cores remain imbalanced
            (dequeue the selected process from the target
             core and enqueue on the core_performing_LB)
46.      release lock on core_performing_LB;
47.      release lock on target_core;
48.    }                    //end of if statement at step no. 40
49.    MAL = MAL+1;
50.  }                      // end of for loop at step no. 4
51.  }                      // end of Algorithm
```

The code given in this algorithm is for periodic as well as *idle load balancing*. When the load balancer will be invoked by any process arriving in the Ready queue from the Wait queue or through fork(), exec() or clone() system calls, a least loaded core will be allocated to that process.

5. SIMULATION AND RESULTS

To evaluate the performance of the Capability Aware Load BalancingAlgorithm, experimentation was done using a simulator of NUMA Multiprocessor/Multicore systems under linux [19], modified by incorporating the proposed algorithm into it. The experimentation was done for different types of NUMA Multicore SystemsAMC1-AMC3having asymmetric cores**,** as following:(i) A**MC1**: 32 core system (16 Nodes, 2 cores per Node, 2MALs)(ii) A**MC2**: 64 coresystem (32 Nodes, 2 cores per Node, 3 MALs)(iii) A**MC3**: 128 core system (32 Nodes, 4 cores per Node, 6 MALs). For each system, different workloads (W1, W2) were generated.

5.1 Results

The results of simulation, in terms of Turn Around Time (TAT) and Performance Gain (improvement in TAT) are given in Tables 1 to 3 and are also depicted in the corresponding graphs given after the respective Tables(**Workload Characteristics** are specified as **W1, W2** in each Table and Graph).

Table 1: Turn Around Time of Processes and Performance Gain for Capability Aware Load Balancing Algorithm vsAsymmetry Unaware Load Balancing Algorithm for NUMA System AMC1

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 (50-300 ms); Arrival: random		
	TAT (ms): Asymmetry UnawareAlgo.	TAT (ms): Capability Aware Algo.	Perf. Gain (%)	TAT (ms): Asymmetry UnawareAlgo.	TAT (ms): Capability Aware Algo.	Perf. Gain (%)
100	248	238	4.03	164	158	3.66
200	419	403	3.82	244	232	4.92
300	603	575	4.64	316	302	4.43
400	786	750	4.58	388	369	4.90
600	1172	1111	5.20	564	537	4.79

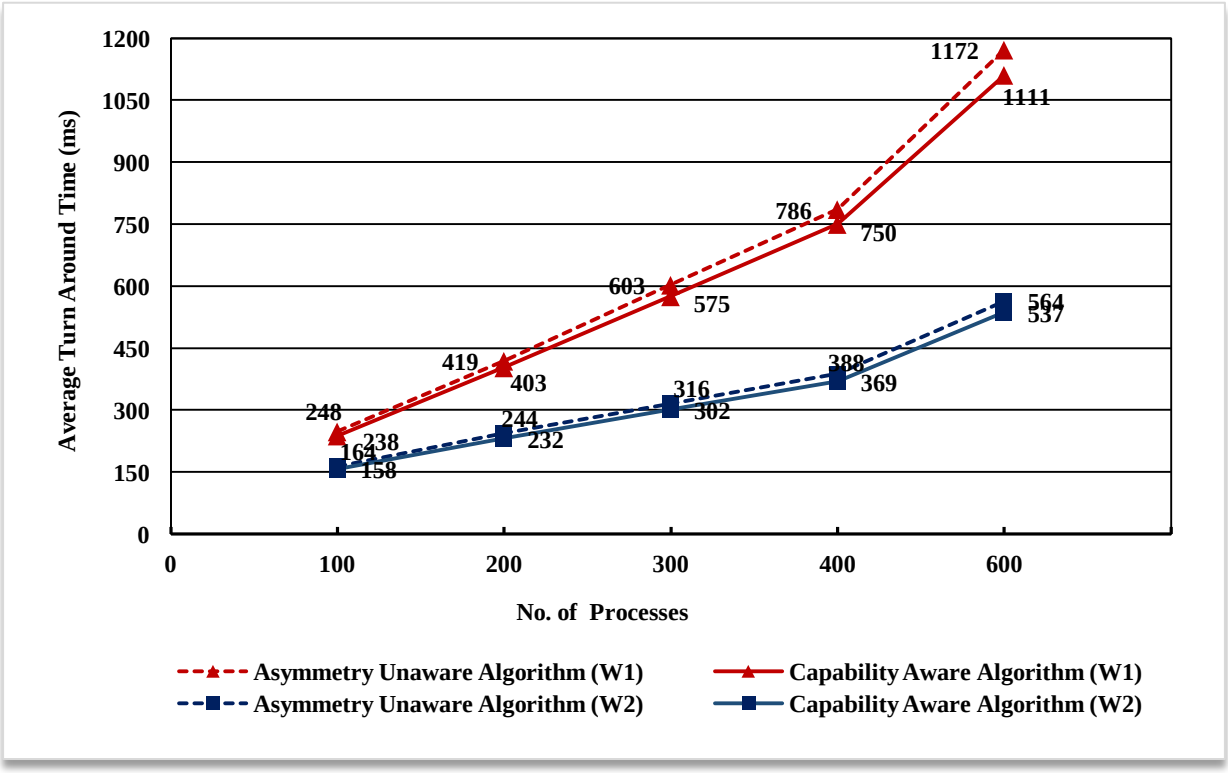

Figure 5: Turn Around Time of Processes for Capability Aware Load Balancing Algorithm vs Asymmetry Unaware Load Balancing Algorithm for NUMA System AMC1

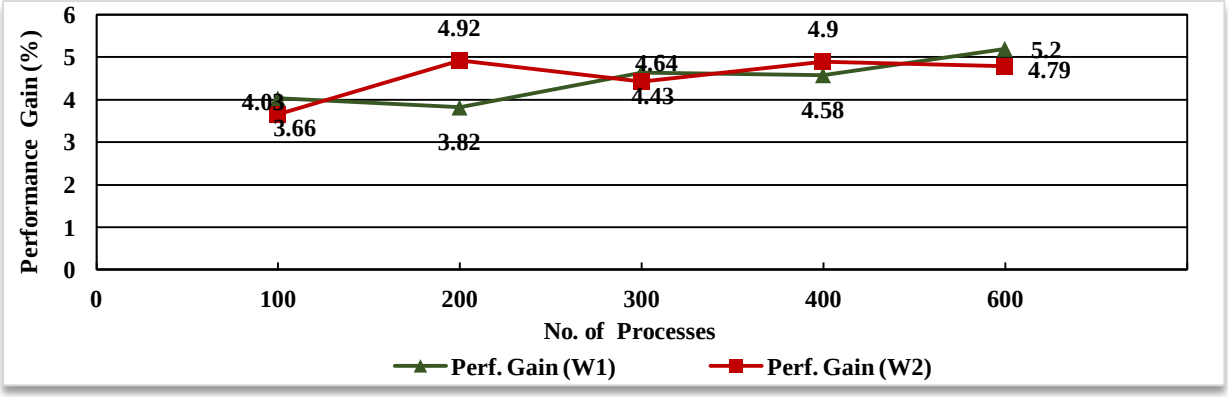

Figure 6: Performance Gain in Capability Aware Load Balancing over Asymmetry Unaware Load Balancing for NUMA System AMC1

Table 2: Turn Around Time of Processes and Performance Gain for Capability Aware Load Balancing Algorithm vs Asymmetry Unaware Load Balancing Algorithm for NUMA System AMC2

No. of processes	W1- Process type: CPU bound; Execu. time: 300 ms; Arrival: same time			W2- Process type: CPU bound; Execu. time: 400 ms ; Arrival: almost same time		
	TAT (ms): Asymmetry Unaware Algo.	TAT (ms): Capability Aware Algo.	Perf. Gain (%)	TAT (ms): Asymmetry Unaware Algo.	TAT (ms): Capability Aware Algo.	Perf. Gain (%)
64	301	289	3.99	384	368	4.17
128	459	434	5.45	660	639	3.18
256	862	827	4.06	1186	1106	6.75
384	1132	1080	4.59	1560	1486	4.74
512	1534	1439	6.19	2141	1993	6.91

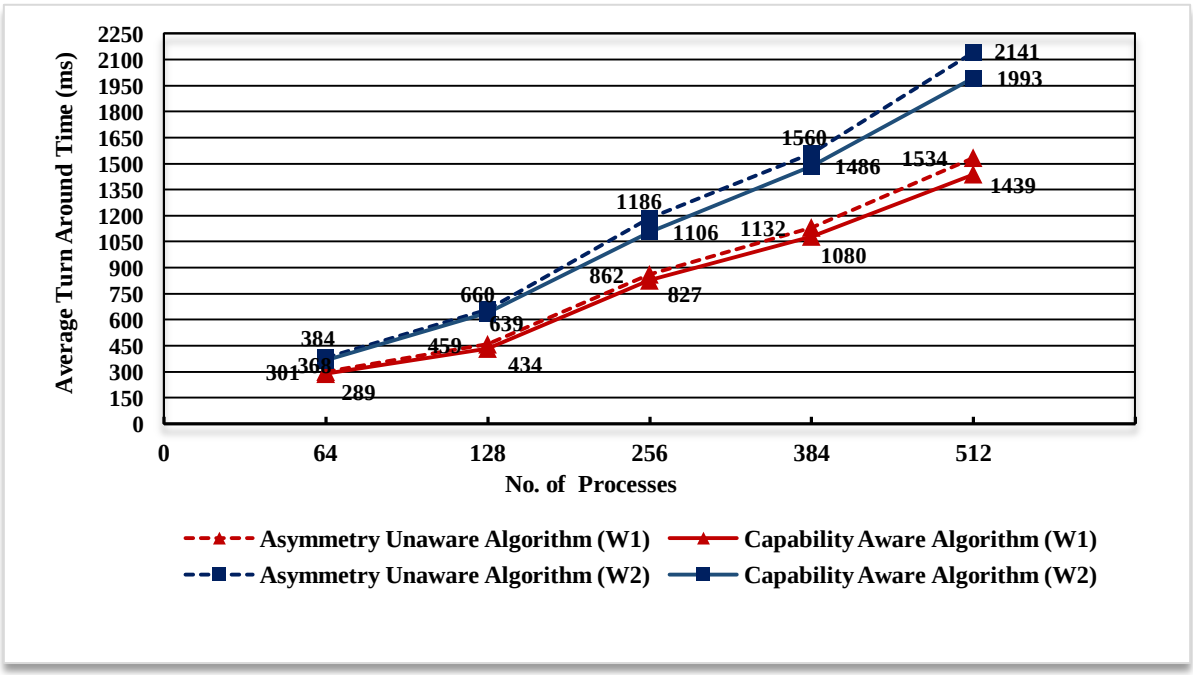

Figure 7: Turn Around Time of Processes for Capability Aware Load Balancing Algorithm vs Asymmetry Unaware Load Balancing Algorithm for NUMA System AMC2

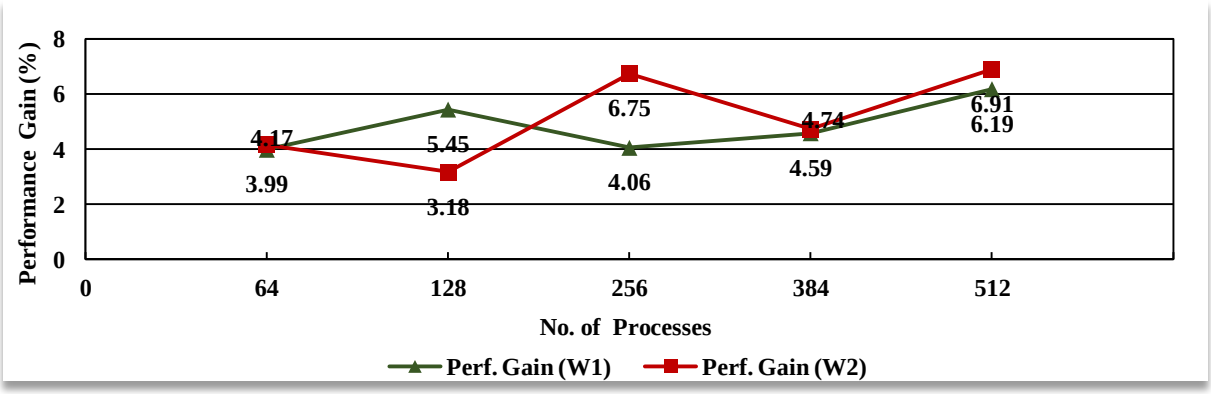

Figure 8: Performance Gain in Capability Aware Load Balancing over Asymmetry Unaware Load Balancing for NUMA System AMC2

Table 3: Turn Around Time of Processes and Performance Gain for Capability Aware Load Balancing Algorithm vsAsymmetry Unaware Load Balancing Algorithm for NUMA System AMC3

No. of processes	W1- Process type: CPU bound; Execu. time: 300 ms; Arrival: almost same time			W2- Process type: Mix of CPU & IO bound; Execu. time: varying (50-400 ms); Arrival: same time		
	TAT (ms): Asymmetry Unaware Algo.	TAT (ms): Capability Aware Algo.	Perf. Gain (%)	TAT (ms): Asymmetry Unaware Algo.	TAT (ms): Capability Aware Algo.	Perf. Gain (%)
100	380	356	6.31	285	274	3.86
200	485	448	7.63	331	316	4.53
300	759	700	7.77	490	457	6.73
400	890	831	6.63	599	561	6.34
500	1204	1093	9.21	730	679	6.98

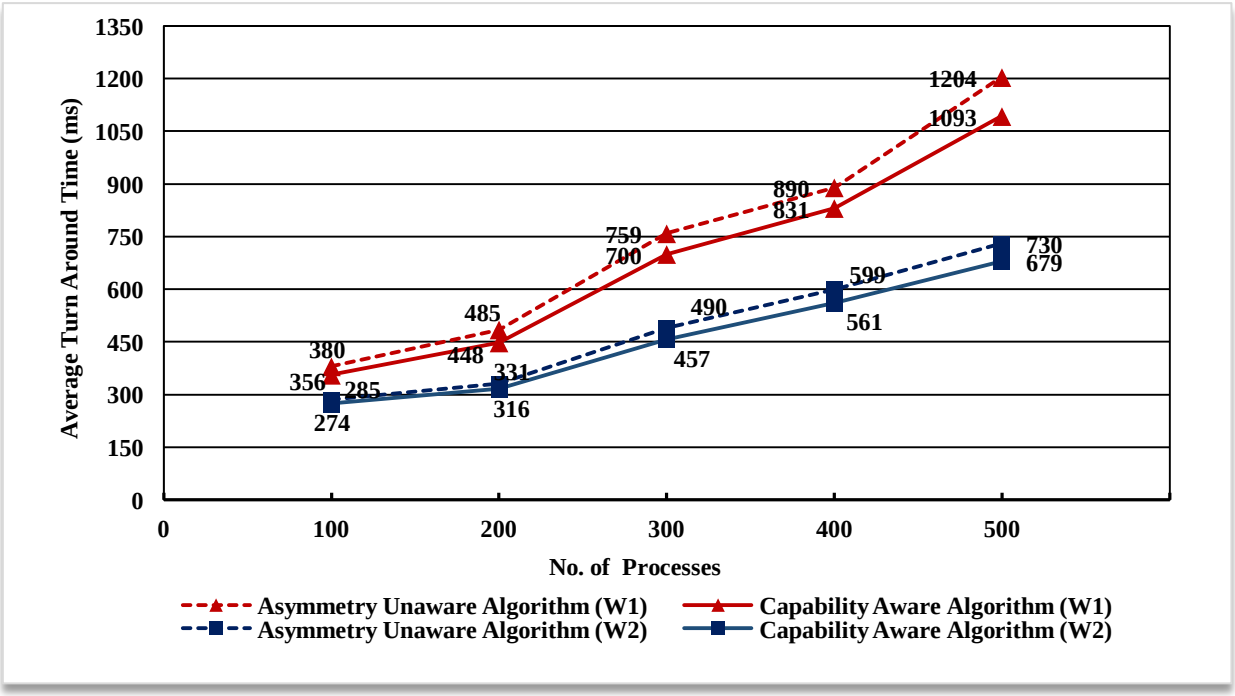

Figure 9: Turn Around Time of Processes for Capability Aware Load Balancing Algorithm vs Asymmetry Unaware Load Balancing Algorithm for NUMA System AMC3

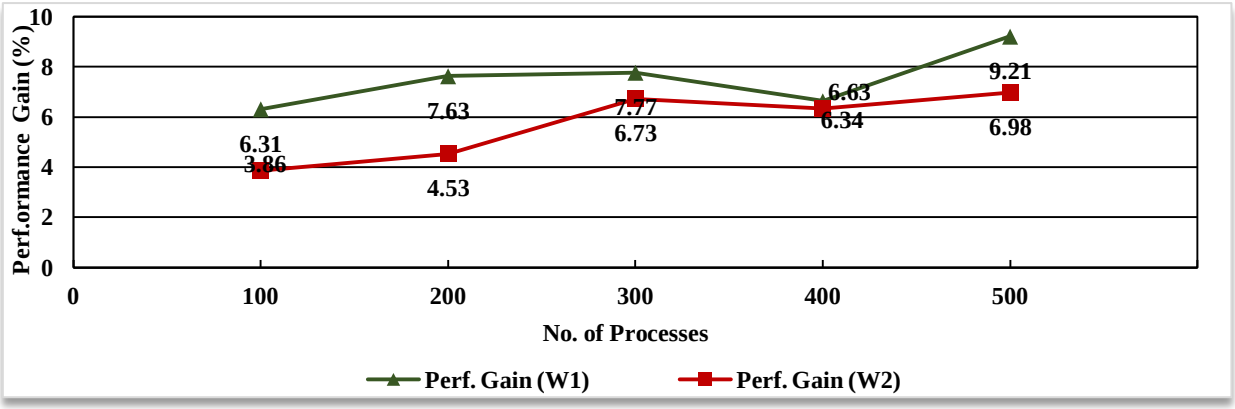

Figure 10: Performance Gain in Capability Aware Load Balancing over Asymmetry Unaware Load Balancing for NUMA System AMC3

5.2 Observations and Discussion on Results

It is evident from the experimentation results that Capability Aware Load Balancing Algorithm outperforms theAsymmetry Unawarelinux load balancing algorithm.For various NUMA systems havingasymmetric cores, it exhibited improved average TAT in the range of 4-9 %. This performance gain achieved, attributes mainly to the reduced *idle load balancing* overheads.

It is further observed that- (i) Forsystems with more no. of cores, performance gain is better as expected. The reason is obvious: with increase in no. of asymmetric cores,unnecessary *idle load balancing*attempts increase in asymmetry unaware load balancing, but not in Capability Aware load balancing(ii) There is variation in performance gain for different sets of processesfor the same system. Thisis due to the difference in arrival time, burst timeand type (CPU bound or IO bound) of the processes.

6. CONCLUSION

One of the factors which hinders the performance improvement of any load balancer is the overheadsarising out of improper load balancing decisions.In the research work presented in this paper, we investigated the reasons of increased overheads in any asymmetry unaware load balancer including that of linux and proposed a Capability Aware Dynamic Load Balancer to achieve the optimum 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 which is therein asymmetry unaware load balancers, and hasenhanced the performance significantly. The work presented in this paper will supplement the efforts of the researchers attempting to design efficient load balancers for upcoming multicore systems, and can be extended for the processors having the other heterogeneity also, apart from the different clock speeds for different cores.

7. REFERENCES

[1] Martin J. Bligh, M. Dobson, D. Hart, and G. Hu Lzenga, "Linux on NUMA Systems," *Linux Symposium*, Vol. 1, 2004, pp. 89-102.

[2] Quan Chen, and Minyi Guo, "Dynamic Load Balancing for Asymmetric Multi-core Architecture," Book Chapter in book: *Task Scheduling for Multi-core and Parallel Architectures*, November 2017, pp 113-151.

[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," *IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, July 2018.

[4] Keng-Mao Cho, Chun-Wei Tsai, Yi-Shiuan Chiu, and Chu-Sing Yang, "A High Performance Load Balance Strategy for Real-Time Multicore Systems," Research Article, *Scientific World Journal Volume 2014*, 14th April 2014.

[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, July 2005.

[6] Alexandra Fedorova, "Operating System Scheduling for Chip Multithreaded Processors," Ph.D. Thesis, Harvard University Cambridge, Massachusetts, September, 2006.

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

[8] S. Hofmeyr, C. Iancu, and F. Blagojevi, "Load Balancing on Speed," Proc. of *15th ACM SIGPLAN Symposiumon Principles and practice of parallel programming (PPoPP '10)*, New York, USA, 2010, pp. 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/, June 10, 2020.

[10] M. Tim Jones, "Inside the Linux Scheduler," Available: www.ibm.com, June 2006.

[11] Changdae Kim, and Jaehyuk Huh, "Exploring the Design Space of Fair Scheduling Supports for Asymmetric Multicore Systems," *IEEE Transactions on Computers* , January 2018, pp(99):1-1.

[12] B. Lepers, V. Qu´ema, and A. Fedorova, "Thread and memory placement on NUMA systems: asymmetry matters", in *Proceedings ofUsenix Annual Technical Conference, USENIX ATC '15*, 2015.

[13] Tong Li, D. Baumberger, D. A. Koufaty, and S. Hahn, ''Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures,'' *Proc. of ACM/IEEE Conf. on Supercomputing*, Nov. 2007, pp. 1–11.

[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*, Dec. 2018.

[16] Robert Love, "Linux Kernel Development," Novell Press, 2nd edition, Jan. 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," *EuroSys '16*, London, UK, April 18 - 21, 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 onSmart Computing and Informatics*, 2019, pp. 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*, Dec. 2013.

[21] L. L. Pilla et al., ''A Hierarchical Approach for Load Balancing on Parallel Multi-Core Systems,'' in *Proc. 41st Int. Conf. Parallel Process* (ICPP), Sep. 2012, pp. 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*, Vol 160, Springer, Singapore, January 2015.

[23] Suresh Siddha, "sched: new sched domain for representing multicore," Available:http://lwn.net/ Articles/ 169277 /

[24] Mukesh Singhal, and Niranjan G. Shivaratri, "Advanced Concepts in Operating Systems," Mcgraw Hill International, 1994.

[25] 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*, Vol. 3, No. 5, October 2011.

[26] Priyesh Kanungo, "Contributions in Dynamic Load Balancing Techniques for Distributed Computing Environment," Ph.D. Thesis, IET-DAVV, Computer Engg., 2007.

[27] Linux Kernel Documentation [Online]. Available:https://www.kernel.org/doc/html/latest/