FACULDADE DE INFORMÁTICA

PUCRS - Brazil

http://www.pucrs.br/inf/pos/

Performance Evaluation of a Multilevel Load Balancing Algorithm

M. Corrêa, R. Chanin, A. Sales, R. Scheer, A. Zorzo

TECHNICAL REPORT SERIES

Number 048
July, 2005

Contact:
mcorrea@inf.pucrs.br

chanin@inf.pucrs.br

asales@inf.pucrs.br

roque.scheer@hp.com

zorzo@inf.pucrs.br

Published by PPGCC - FACIN - PUCRS
Av. Ipiranga, 6681
90619-900 Porto Algre - RS - Brasil

1 Introduction

The demand for computational power has been increasing throughout the past years. Several solutions are being used in order to supply such demand, *e.g.*, clusters of workstations [6] and shared memory multiprocessors. Although clusters have lower cost, their use implies in a great specialized programming effort. It is necessary to build new applications or port the existing ones to execute in these environments through the use of specific API's, such as MPI (*Message Passing Interface*) [23]. On the other hand, shared memory multiprocessor computers are more expensive, but simpler to use, since all resources are managed by a single operating system.

Shared memory multiprocessor computers can be classified as UMA (*Uniform Memory Access*) or NUMA (*Non-Uniform Memory Access*) computers [16]. In UMA computers each processor can access any memory area with the same average cost. This property simplifies the load balancing: a process can be moved to any processor without any impact to its memory access time, if the process is not *cache-hot*. The major drawback of UMA architectures is that the number of processors is limited by the contention on access to memory, which becames a bottleneck, since they all share the same memory bus. NUMA architectures allow a greater number of processors because processors and memory are distributed in nodes. Memory access times depend on the processor that a process is executing and on the accessed memory area. Thus, the load balancing on these machines is more complex, since moving a process to a node that is distant from its memory area can increase process execution time.

Load balancing for parallel environments is a problem that has been deeply studied for a long time. However, most of these studies are focused on user-level load balancing, where users of the system must know their applications behavior and provide information to the load balancing algorithm. In this sense, there are many proposals for different platforms, for example clusters [2, 5, 26] and computational grids [13, 24]. Some authors have also presented solutions or studies for the load balancing problem on NUMA computers. Zhu [25], for instance, proposes a cluster queue structure for processes based on a hierarchical structure in order to solve the limitation of single queue systems and the load imbalance problem that results of distributed queues. Focht [12], on the other hand, describes an algorithm based on the Linux load balancing algorithm that tries to attract processes back to their original nodes when they are migrated. There has been also some studies presenting analysis of load balancing algorithms on NUMA machines [7].

In [10] we proposed an algorithm that allows Linux to perform multilevel load balancing in NUMA computers. The current Linux load balancing algorithm uses a structure called *sched domain* to build a hierarchy that represents the machine's

topology. Based on this hierarchy, Linux tries to keep processes closer to their memory areas, moving them among processors in the same node before performing inter-node migration [4]. The algorithm that is responsible for constructing this hierarchy assumes that all NUMA machines have only two memory access levels. However, there are NUMA computers with more than two memory access levels. The hierarchy built for these machines, therefore, does not represent their topology correctly, causing unappropriate load balancing. To cope with this problem, we proposed a generic algorithm to build a multilevel *sched domain* hierarchy, according to the number of memory access levels that the NUMA computer contains.

In this paper we evaluate the performance of the Linux load balancing algorithm in different NUMA architectures, using the 2-level *sched domain* hierarchy built by the current Linux version and using an *n*-level hierarchy built by our proposed algorithm. To perform this evaluation we use two different approaches: *simulation* and *analytical* models. The simulation model is developed using the *JavaSim* simulation tool [17], and the analytical model is described using the *Stochastic Automata Networks* (SAN) formalism [22].

This paper is organized as follows. Section 2 describes the current Linux load balancing algorithm and our proposal to allow Linux to perform multilevel load balancing. Section 3 describes the implementation of our proposal. Sections 4 and 5 present the simulation and analytical models respectively, which were used to compare Linux load balancing performance using the current algorithm and the proposed solution. Section 6 shows the results of both evaluation models and demonstrate that multilevel load balancing can present a better performance in terms of average processes execution time than the current Linux load balancing algorithm. Finally, Section 7 assesses future work and emphasizes the main contributions of this paper.

2 Load Balancing in NUMA Computers

In a NUMA computer, processors and main memory are distributed in nodes. Each processor can access the entire memory address space, but with different latency times [16]. In general, if the system has a small number of processors, the machine has only two memory access levels. For example, Figure 1 shows the architecture of a HP Integrity Superdome server [14] with 4 nodes and 16 processors. this machine has two different memory latencies: when a processor accesses memory that is inside its node; and when a processor accesses any other memory area.

However, some NUMA architectures, usually with a greater number of processors, have more than two memory access levels, *e.g.*, the HP Integrity Superdome

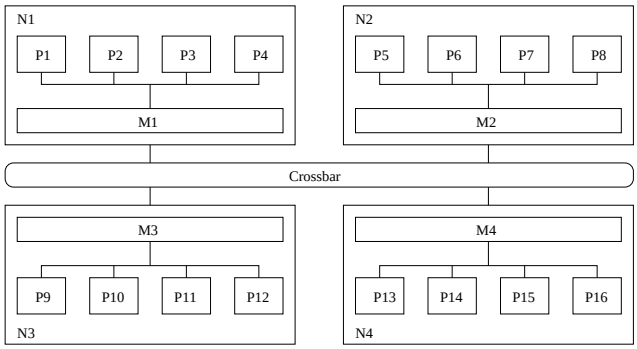


Figure 1: HP Integrity Superdome with two memory access levels

server (Figure 2) and the SGI Altix 3000 servers [20]. The machine shown in Figure 2 is a NUMA computer with 16 nodes, 64 processors and three memory access levels: (i) memory latency inside the node; (ii) memory latency among nodes on the same crossbar; and (iii) memory latency among nodes on different crossbars.

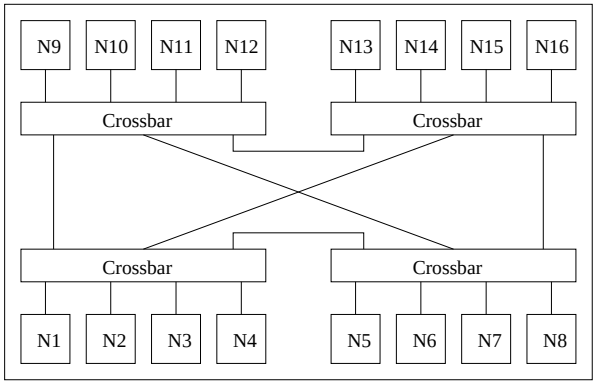


Figure 2: HP Integrity Superdome with three memory access levels

An efficient load balancing algorithm must be aware of how many memory access levels exist in the machine in order to keep processes as close as possible to their memory areas, improving their average execution time.

2.1 Linux Load Balancing Algorithm

Up to kernel version 2.4, the Linux process scheduler had a single shared process queue. When a processor was idle, it received a process from this queue. Although

this approach results in a natural load balancing, it was not scalable: as the number of processors increased, the process queue became a bottleneck. The current Linux scheduler has one process queue for each processor in the system, solving the scalability problem. However, due to the multiple process queues, Linux had to implement a load balancing algorithm [19].

The Linux load balancing algorithm uses a data structure called *sched domain* to build a hierarchy that represents the machine's topology. Each *sched domain* contains CPU groups that define the scope of load balancing to this domain [1, 4]. Based on this hierarchy, Linux is able to perform appropriate load balancing to different architectures. For NUMA machines, Linux builds a two-level *sched domain* hierarchy: the lowest level is composed of processors that are in the same node and the highest level contains all processors of the system. Thus, for the machine in Figure 1, Linux builds the hierarchy shown in Figure 3.

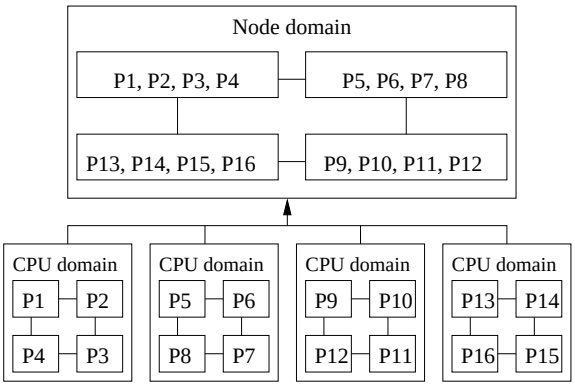


Figure 3: 2-level *sched domains* hierarchy

Load balancing is performed among processors of a specific *sched domain*. Since load balancing must be executed on a specific processor (all processors will execute the load balancing eventually), it will be performed in the *sched domains* that contain this processor. Initially, the load balancer searches for the busiest processor of the busiest CPU group in the current domain (all domains are visited, starting at the lowest level). Then, if the busiest processor is overloaded in comparison to the processor that is executing the load balancing, Linux migrates processes from the overloaded processor to the processor that is executing the load balancing.

2.2 Multilevel Load Balancing Algorithm

The load balancing algorithm described in Section 2.1 tries to keep processes closer to their memory areas. The memory area of a process is allocated in the same node of the processor in which this process was created. Thus, Linux migrates tasks among processors in the same node, keeping the processes closer to their memory areas. If after this intra-node migration there is still an imbalance, processes will be moved from distant nodes.

However, in machines with more than two memory access levels, there are different distances among nodes. If inter-node migration is necessary, it is desirable to move processes among closer nodes, performing a multilevel load balancing. Linux load balancing does not implement this feature, because it is not aware of the different node distances. This happens because the created *sched domains* hierarchy does not represent correctly the machine's topology if the computer has more than two memory access levels: all processors in different nodes are grouped in only one *sched domain* (the highest level of the hierarchy).

In order to solve this problem, we have proposed a new algorithm to build the *sched domains* hierarchy [10]. This is a generic algorithm that builds an *n*-level hierarchy for a machine with *n* memory access levels, based on node distances provided by the ACPI SLIT table [15]. Hence, for the machine in Figure 2, which has three memory access levels, our proposed algorithm builds the *sched domains* hierarchy shown in Figure 4.

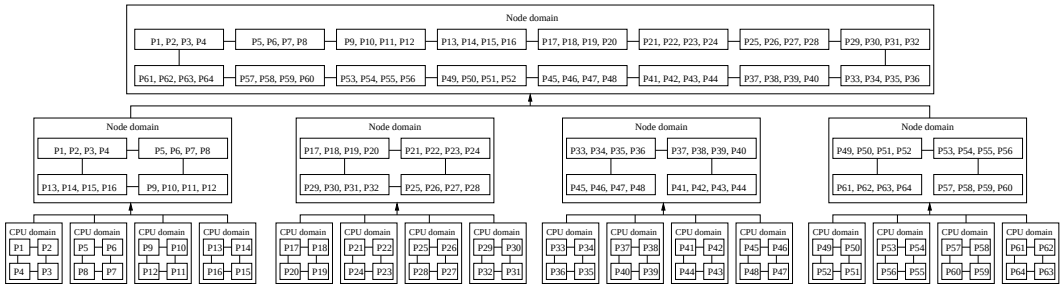


Figure 4: 3-level *sched domains* hierarchy

Using the hierarchy from Figure 4, after the intra-node migration, Linux performs load balancing in the second level of the hierarchy, moving processes among closer nodes, *i.e.*, on the same crossbar. The processes will be migrated to the most distant nodes in the third level of the hierarchy, only if there is still load imbalance among nodes. Thus, with the 3-level *sched domain* hierarchy, Linux can actually try to keep the processes closer to their memory area, improving their execution times.

3 Implementation

In the current Linux kernel version, the *sched domains* hierarchy is built in the *arch_init_sched_domains* function, located in the *kernel/sched.c* file. Since Linux builds a two-level hierarchy for any NUMA machine (considering that there are no SMT (*Simultaneous Multithreading*) processors), all *sched domains* and CPU groups are declared previously. In the *arch_init_sched_domains* function, the domains are configured for each processor according to the following order: first the node domain, which groups all processors of the system; next the CPU domain, which groups the processors in the same node. The CPU groups configuration, which are shared for all processors, is performed after the *sched domains* configuration.

In order to implement the proposed algorithm for the construction of a multilevel *sched domains* hierarchy described in [10], we have changed this specific function and created some auxiliary functions. In our approach, the number of node domains depends on memory access levels. Therefore, we use the previously declared node domain for the lowest level and allocate memory to create all other domains and their CPU groups dynamically.

Code 1 shows node domains creation for a processor *i*. The `base` *sched domain* is the lowest level node domain (line 2). The `nodemask` and `cpu_map` variables are bit sequences with one bit for each processor in the system. The `cpu_map` sequence represents all available processors. The `nodemask` sequence indicates processors that were inserted in the last created *sched domain*. Before node domains creation, our modified function configures the CPU domain (`sd` variable). The bits of the processors in the same node of processor `i` are set in the `nodemask` variable (due to the CPU domain contains these processors). Thus, the code inside the `while` command will be executed until all processors are inserted in the last created *sched domain*, which will be the higher level of the hierarchy.

The `find_next_best_nodes` (line 4) function uses node distances provided by the ACPI SLIT table to determine the processors that will be inserted in the next *sched domain*. This function sets the bits of these processors in the `nodemask` variable, which defines the processors (`span`) of the new *sched domain*. If the *sched domain* that will be configured is the first node domain, it uses the memory area that was allocated previously (lines 7 to 10). Otherwise, a new memory area is allocated (lines 11 to 14) for this node domain.

Code 1 Node domains hierarchy creation

```
1  #ifdef CONFIG_NUMA
2    struct sched_domain *base = NULL;
3    while(!cpus_equal(nodemask, cpu_map)) {
4      find_next_best_nodes(node, &nodemask);
5      cpus_and(nodemask, nodemask, cpu_map);
6      p = sd;
7      if (base == NULL) {
8        sd = &per_cpu(node_domains, i);
9        base = sd;
10     }
11     else {
12       sd = (struct sched_domain *)kmalloc(
13        sizeof(struct sched_domain),GFP_KERNEL);
14     }
15     *sd = SD_NODE_INIT;
16     sd->span = nodemask;
17     sd->groups = NULL;
18     p->parent = sd;
19     sd->parent = NULL;
20   }
21 #endif
```

The created *sched domain* is configured in lines 15 to 19. The parameters that are common to all node domains are defined in `SD_NODE_INIT` (line 15). The hierarchical organization is built by the *sched domain* `parent` variable, which points to the last created domain (line 18).

Based on this implementation, we have developed a patch for the current Linux kernel version. This patch is available in the PeSO project web site [21]. Benchmark results showed that our solution does not introduce any kind of overhead for the load balancing in the HP Superdome computer from Figure 1.

4 Simulation Model

In order to compare the performance of the Linux load balancing algorithm when the *sched domains* hierarchy is built using the current Linux algorithm and using our proposed algorithm, we have implemented a simulation model using the *JavaSim* simulation tool [17].

JavaSim is a Java implementation of the *C++SIM* [11, 18] simulation toolkit, which supports the continuous time-discrete event simulation model. Event schedul-

ing is process-oriented, *i.e.*, the management of events is implicit in the management of processes [3]. Specific system models can be constructed through inheritance of *JavaSim* classes.

Our modeled system simulates the Linux process scheduling and load balancing algorithms for different architectures and workloads. We have three classes that implement simulation processes: *NumaMachine, Arrivals* and *Processor*. The model contains other classes that are used by the simulation processes, for example, *Task* class.

In a given simulation, there is only one instance of *NumaMachine*, which contains information about the machine's topology. The *NumaMachine* object controls the simulation execution and the creation and initial activation of all other simulation processes.

There is also only one instance of *Arrivals*. This object is responsible for creating tasks and assigning them to processors, according to Linux policies. The task creation rate is defined by an exponential distribution with variable mean.

Processor objects are the main simulation processes. Since Linux scheduler executes in an independent way in each processor of the system, both scheduling and load balancing algorithms are executed by these objects.

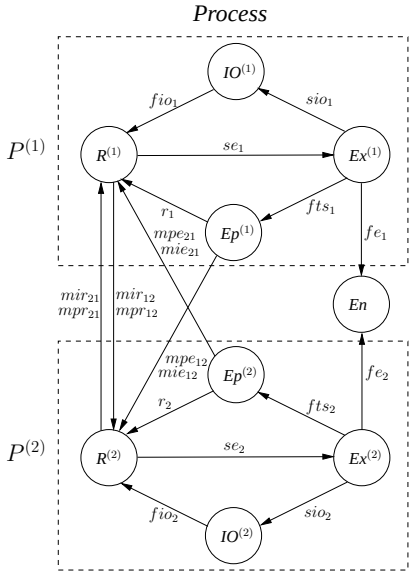
Task objects represent Linux processes. Their nice values (static priority) are determined according to uniform distribution from -20 to 20. Dynamic priority and timeslice values are calculated based on the nice value. The execution time of a *Task* object is also defined by an exponential distribution with variable mean. The value of execution time is defined by the time that this task would execute if it was not interrupted. Similar to Linux, tasks are not classified as totally *CPU-bound* or *I/O-bound*. They can be more or less CPU or I/O-bound, depending on the time they spent executing (processing time) and the time they waited for an I/O operation to finish (waiting time). Thus, the time that a task will execute without losing the processor is defined by an exponential distribution and it is always less than or equal to the timeslice. If a task executes for less than its timeslice, it means that the task yielded the processor to wait for some I/O operation. The time the task will be in a waiting state is also defined by an exponential distribution.

The following values must be provided in order to start the simulation: (i) node distances, which define the number of memory access levels; (ii) number of tasks that will be created and executed by the processors; (iii) number of processors and nodes; (iv) load balancing type, which defines if the simulation will use the current Linux algorithm or the proposed one to build the *sched domain* hierarchy and to perform the load balancing; (v) average tasks creation rate; (vi) average tasks execution time; (vii) average tasks processing time; and (viii) average tasks waiting time.

The simulation model is flexible enough to provide many different results. In this paper we show as results the average execution time of all created processes.

5 Analytical Model

The main idea of our analytical model is to describe the behavior of only one process in the Linux system, but considering the influence of other processes. We propose a system model consisting of P processors and only one process.

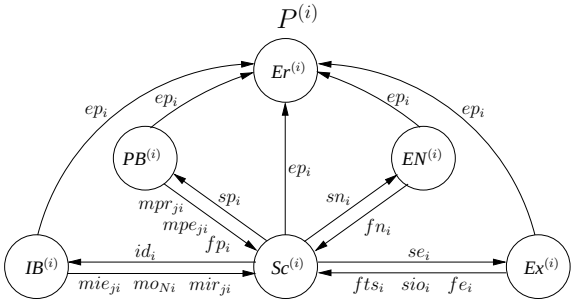


Type	Event	Rate
syn	sio_1	r_{sio_1}
syn	fts_1	r_{fts_1}
syn	mpe_{12}	$r_{mpe_{12}}$
loc	fio_1	r_{fio_1}
loc	r_1	r_{r_1}
syn	mie_{12}	$r_{mie_{12}}$
syn	se_1	r_{se_1}
syn	fe_1	r_{fe_1}
syn	mpr_{12}	$r_{mpr_{12}}$
syn	mir_{12}	$r_{mir_{12}}$
syn	sio_2	r_{sio_2}
syn	fts_2	r_{fts_2}
syn	mpe_{21}	$r_{mpe_{21}}$
loc	fio_2	r_{fio_2}
loc	r_2	r_{r_2}
syn	mie_{21}	$r_{mie_{21}}$
syn	se_2	r_{se_2}
syn	fe_2	r_{fe_2}
syn	mpr_{21}	$r_{mpr_{21}}$
syn	mir_{21}	$r_{mir_{21}}$

Figure 5: Automaton *Process*

Figure 5 shows the SAN model of a process in a 2-processor machine and its transition rate table. The *Process* automaton is composed of the following states: $R^{(i)}$ representing that the process is in the ready queue (waiting to be scheduled) in the i^{th} processor; $Ep^{(i)}$ representing that the process is in the expired queue (it has finished its timeslice and is waiting to be "moved" to the ready queue) in the i^{th} processor; $Ex^{(i)}$ representing the situation in which the process is executing in the corresponding processor; $IO^{(i)}$ representing the situation in which the process is waiting for an input/output operation; and *En* representing that the process has finished its execution.

In Figure 5, the process can execute in only two processors ($P^{(1)}$ and $P^{(2)}$). In order to represent a greater number of processors it is necessary to replicate states $R^{(i)}$, $IO^{(i)}$, $Ex^{(i)}$ and $Ep^{(i)}$ and their corresponding transitions.



Type	Event	Rate	Type	Event	Rate	Type	Event	Rate
syn	mie_{ji}	$r_{mie_{ji}}$	loc	mo_{Ni}	$r_{mo_{Ni}}$	syn	mir_{ji}	$r_{mir_{ji}}$
syn	mpr_{ji}	$r_{mpr_{ji}}$	syn	mpe_{ji}	$r_{mpe_{ji}}$	loc	fp_i	r_{fp_i}
loc	id_i	f_{id_i}	loc	sp_i	r_{sp_i}	loc	ep_i	r_{ep_i}
loc	sn_i	r_{sn_i}	loc	fn_i	r_{fn_i}	syn	se_i	r_{se_i}
syn	fts_i	r_{fts_i}	syn	sio_i	r_{sio_i}	syn	fe_i	r_{fe_i}

$$f_{id_i} = \left[(st\ \textit{Process}\ != R^{(i)})\ \&\&\ (st\ \textit{Process}\ != Ep^{(i)})\right] * r_{id_i}$$

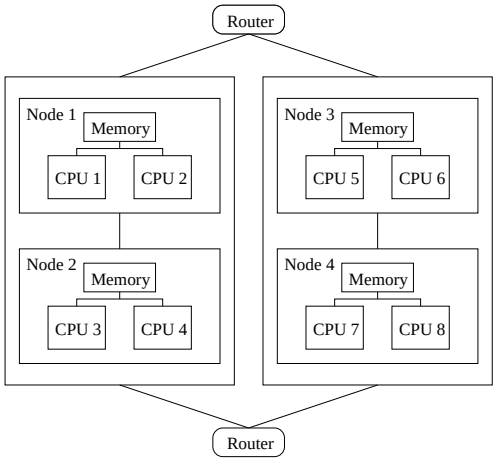
Figure 6: Automaton *Processor*

Figure 6 shows the processors modeled using SAN and the corresponding transition rate table. A processor might be in one of the following states: $IB^{(i)}$ representing that the processor is not being used and it is performing the load balancing algorithm; $Sc^{(i)}$ representing that the processor is executing the scheduling algorithm; $PB^{(i)}$ representing that the processor is executing the periodical load balancing algorithm; $EN^{(i)}$ representing that the processor is executing any other process; $Ex^{(i)}$ representing that the processor is executing the process showed in Figure 5; and $Er^{(i)}$ representing that some error has occurred and the processor is not working. More details can be found in [8].

6 Numerical Results

The example machine used in our analytical test (Figure 7), which is based on SGI Altix architectures, has four nodes, eight processors and four memory access levels. As mentioned in Section 5 our analytical model considers the behavior of one specific process.

Figure 8 presents analytical results for the machine in Figure 7, considering a workload of 200 processes. The results show the normal end probability of a high-priority I/O-bound process *P* when Linux recognizes only two memory access levels (*4 memory access levels - 2 Linux sched domain levels*) and when Linux recognizes the actual computer's topology (*4 memory access levels - 4 Linux sched domain*

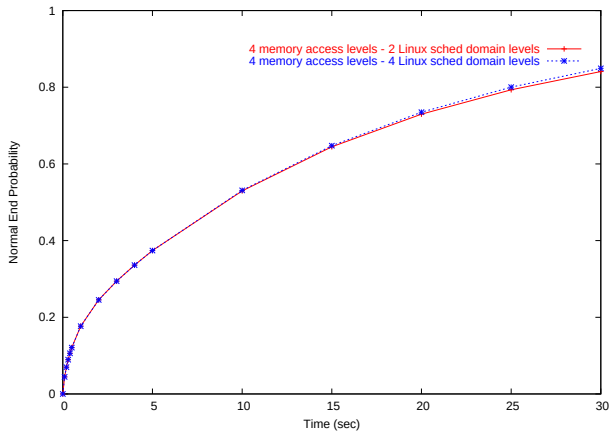


sched domains for CPU 1

Level	Processors
1	1,2
2	1,2,3,4
3	1,2,3,4,5,6
4	1,2,3,4,5,6,7,8

Figure 7: NUMA machine, 4 memory access levels

levels). According to the results, after 30 seconds the probability that process *P* finishes its execution is greater when our proposal is used (improvement of 1%).

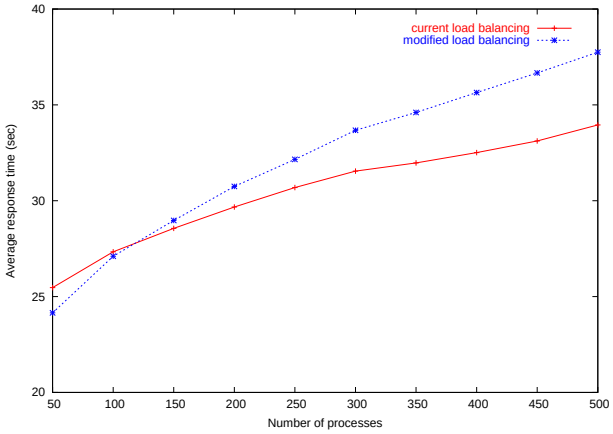


Time	4 levels 2 sched	4 levels 4 sched	Time	4 levels 2 sched	4 levels 4 sched
0	0.00000	0.00000	3	0.29422	0.29416
0.1	0.04438	0.04438	4	0.33600	0.33599
0.2	0.06987	0.06987	5	0.37399	0.37407
0.3	0.08937	0.08937	10	0.52968	0.53098
0.4	0.10608	0.10608	15	0.64455	0.64768
0.5	0.12091	0.12091	20	0.72988	0.73495
1	0.17688	0.17687	25	0.79346	0.80027
2	0.24544	0.24538	30	0.84099	0.84921

Figure 8: Analytical results

Regarding the simulation results, we have defined an environment similar to the one considered in the analytical model[1]: nine processes with nice values equal to 0 and all other processes with smaller priorities (nice values from 1 to 19), average execution time of 500 miliseconds, average processing time of $timeslice+nice\_value$ miliseconds and average waiting time of 1 second. In our simulation tests we have defined workloads between 50 and 500 processes. For each test case, we performed 1,000 simulation runs. The simulation results shown in this section represent the average of the results obtained from these runs.

In the first simulation test we considered the same machine used in the analytical model (Figure 7). Figure 9 shows the average processes execution time on this machine with different workloads. According to the results, when the number of processes increases, the system performance decreases when our algorithm is used. This situation occurs because there are few processors per node. When the number of processes increases, processes will be migrated to distant nodes and the probability they return back to their original nodes is smaller than when the current algorithm is used. This situation could be solved if memory pages were also migrated, reducing memory access time of processes [9].

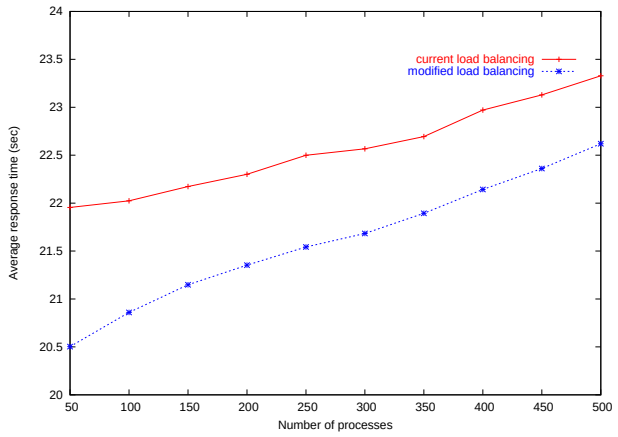


# processes	current	modified	# processes	current	modified
50	25.467	24.152	300	31.544	33.672
100	27.337	27.103	350	31.970	34.605
150	28.558	28.973	400	32.513	35.642
200	29.671	30.747	450	33.116	36.667
250	30.685	32.158	500	33.955	37.748

Figure 9: 4 levels, 8 processors

[1]Some of the values are based on actual benchmark results or Linux constants.

If we consider the same machine, but with four processors per node, the simulation results show that, for all tested workloads, the execution time of processes decreases when load balancing is performed using the 4-level *sched domains* hierarchy, which represents the machine's topology correctly (improvement of 4%). This result is presented in Figure 10.



# processes	current	modified	# processes	current	modified
50	21.955	20.503	300	22.567	21.684
100	22.024	20.860	350	22.694	21.894
150	22.173	21.148	400	22.972	22.143
200	22.301	21.353	450	23.129	22.361
250	22.499	21.543	500	23.329	22.620

Figure 10: 4 levels, 16 processors

We have also considered some comercial machines in the simulations to verify the system performance when our proposed algorithm is used. In order to compare the results shown above with the peformance of actual machines, we considered two different computers: the SGI Altix 3000 server with six memory access levels (Figure 11) and the HP Integrity Superdome computer with three memory access levels (Figure 2).

The SGI Altix computer has 16 nodes and only two processors per node. The simulation results for this machine are presented in Figure 12. For this machine, the average performance improvement was 3%, but for a workload of 300 or more processes the performance of our proposal is worse than the performance of the current algorithm. This is the same situation shown in Figure 9, since both computers have several memory access levels and few processors per node.

On the other hand, the HP Superdome computer in Figure 2 has only three memory access levels and four processors per node. For this machine, the simu-

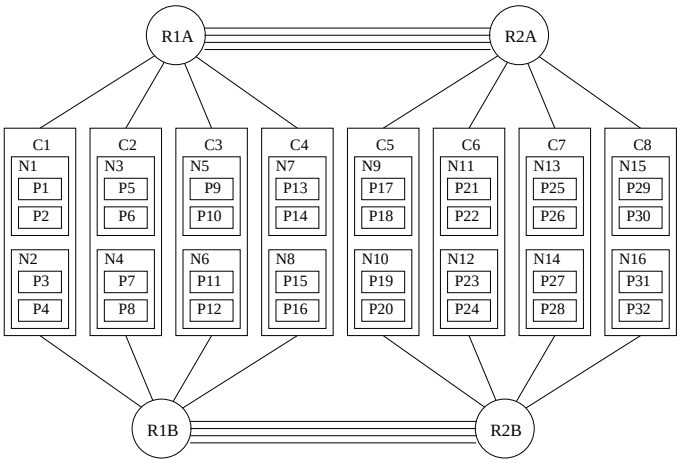
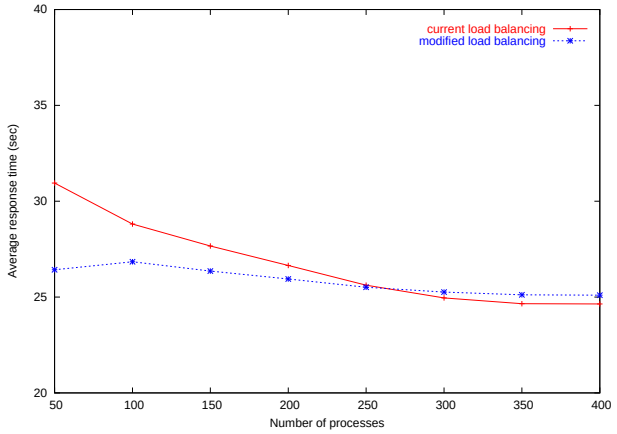


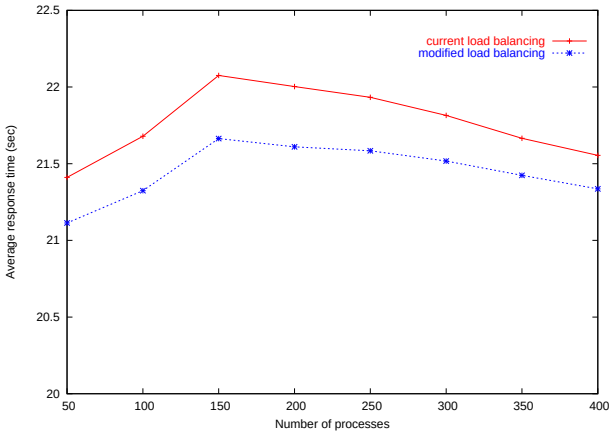
Figure 11: SGI Altix 3000 server, 6 memory access levels



# processes	current	modified	# processes	current	modified
50	30.949	26.424	250	25.621	25.513
100	28.807	26.843	300	24.955	25.259
150	27.666	26.355	350	24.656	25.115
200	26.649	25.943	400	24.642	25.101

Figure 12: SGI Altix 3000 simulation results

lation achieves an improvement for all tested workloads (Figure 13). The average performance improvement obtained for this machine was 1.5%.



# processes	current	modified	# processes	current	modified
50	21.410	21.113	250	21.933	21.584
100	21.679	21.324	300	21.815	21.517
150	22.076	21.664	350	21.665	21.424
200	22.003	21.609	400	21.554	21.336

Figure 13: HP Superdome simulation results

7 Conclusion

This paper has discussed the performance benefits of a new approach for load balancing in NUMA computers. In order to evaluate this new approach, we have developed an analytical and a simulation model. The results we obtained from these models have driven an actual implementation of our approach in the Linux operating system. Linux does implement a load balancing algorithm but its implementation considers only NUMA computers that have up to two memory access levels. Such implementation does not take full advantage of the correct machine's topology, as shown in our analytical and simulation results.

The actual architectures we have used in our models were the HP Superdome and SGI Altix 3000 with different number of memory access levels. We have also implemented our strategy on the HP Superdome Computers, and the results we obtained are very promissing.

The major contributions of this paper are related to the use of the ACPI SLIT table information to build the *sched domains* hierarchy; analytical and simulation results for a load balancing algorithm in NUMA computers; importance of the correct use of machine's topology during the load balancing; and actual implementation of our proposal on an actual operating system.

From our simulation model we have already verified that in at least one situation

our solution would not improve the overall system performance. This situation occurs when a machine has several memory access levels and few processors per node. Such problem is due to the fact that Linux does not migrate process memory when the process is migrated. We have already constructed an analytical model showing the benefits of memory page migration in NUMA computers [9]. Our next step is to integrate both Linux load balancing and memory migration.

References

[1] J. Aas. Understanding the Linux 2.6.8.1 CPU Scheduler. Available at URL http://josh.trancesoftware.com/linux.

[2] G. Attiya and Y. Hamam. Two Phase Algorithm for Load Balancing in Heterogeneous Distributed Systems. In *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 434–439, 2004.

[3] J. Banks, J. Carson II, B. Nelson, and D. Nicol. *Discrete-Event System Simulation*. Prentice-Hall, 2001.

[4] M. J. Bligh, M. Dobson, D. Hart, and G. Huizenga. Linux on NUMA Systems. In *Proceedings of the Linux Symposium*, pages 89–102, July.

[5] C. Bohn and G. Lamont. Load Balancing for Heterogeneous Clusters of PCs. *Future Generation Computer Systems*, 18:389–400, 2002.

[6] R. Buyya. *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall, 1999.

[7] M. Chang and H. Kim. Performance Analysis of a CC-NUMA Operating System. In *15th International Parallel and Distributed Processing Symposium*, 2001.

[8] R. Chanin, M. Corrêa, P. Fernandes, A. Sales, R. Scheer, and A. Zorzo. Analytical Modeling for Operating System Schedulers on NUMA Systems. *Eletronic Notes in Theoretical Computer Science*, Accepted for publication, 2005.

[9] R. Chanin, M. Corrêa, A. Sales, G. Tesser, and A. Zorzo. Memory Page Migration on NUMA Systems:An Analytical Model. In *10th Asia-Pacific Computer Systems Architecture Conference*, October 2005. (Submitted).

[10] M. Corrêa, R. Chanin, A. Sales, R. Scheer, and A. Zorzo. Multilevel Load Balancing in NUMA Computers. Technical Report TR 049, PUCRS, Porto Alegre, 2005.

[11] C++SIM Web Site. http://cxxsim.ncl.ac.uk.

[12] E. Focht. NUMA aware scheduler. Available at URL http://home.arcor.de/efocht/sched, 2002.

[13] D. Grosu and A. Chronopoulos. Algorithmic Mechanism Design for Load Balancing in Distributed Systems. *IEEE Transactions on Systems, Man and Cybernetics*, 34(1), 2004.

[14] Hewlett-Packard. Meet the HP Integrity Superdome. Available at URL http://h21007.www2.hp.com/dspp/files/unprotected/superdomejan05.pdf, 2005.

[15] Hewlett-Packard, Intel, Microsoft, Phoenix and Toshiba. Advanced Configuration and Power Interface Specification. Available at URL http://www.acpi.info/DOWNLOADS/ ACPIspec30.pdf, 2004.

[16] K. Hwang and Z. Xu. *Scalable Parallel Computing - Technology, Architecture and Programming*. WCB/McGraw-Hill, 1998.

[17] JavaSim Web Site. http://javasim.ncl.ac.uk.

[18] M. C. Little and D. L. McCue. Construction and Use of a Simulation Package in C++. *C User's Journal*, 12(3), 1994.

[19] R. Love. *Linux Kernel Development*. SAMS, Developer Library Series, 2003.

[20] M. Woodacre and D. Robb and D. Roe and K. Feind. The SGI® AltixTM 3000 Global Shared-Memory Architecture. Available at URL http://www.sgi.com/products/servers/altix/whitepapers, 2005.

[21] PeSO Project Web Site. http://www.inf.pucrs.br/peso.

[22] B. Plateau and K. Atif. Stochastic Automata Networks for modelling parallel systems. *IEEE Transactions on Software Engineering*, 17(10):1093–1108, 1991.

[23] M. Snir, S. Otto, S. Huss-Lederman, D.W. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, 1996.

[24] Y. Wang and J. Liu. Macroscopic Model of Agent-Based Load Balancing on Grids. In *2nd International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 804–811, 2001.

[25] W. Zhu. Cluster Queue Structure for Shared-Memory Multiprocessor Systems. *The Journal of Supercomputing*, 25:215–236, 2003.

[26] Y. Zhuang, T. Liang, C. Shieh, J. Lee, and L. Yang. A Group-Based Load Balance Scheme for Software Distributed Shared Memory Systems. *The Journal of Supercomputing*, 28:295–309, 2004.