FACULDADE DE INFORMÁTICA
PUCRS - Brazil

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

Multilevel Load Balancing in NUMA Computers

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

Technical Report Series

Number 049
July, 2005

Contact:
mcorrea@inf.pucrs.br

chanin@inf.pucrs.br

asales@inf.pucrs.br

roque.scheer@hp.com

zorzo@inf.pucrs.br

Copyright © Faculdade de Informática - PUCRS
Published by PPGCC - FACIN - PUCRS
Av. Ipiranga, 6681
90619-900 Porto Algre - RS - Brasil

1 Introduction

The demand for computing power and resources has being increasing throughout the past years. Single processor machines are still being used, but it is becoming quite common to find multiprocessor machines even in ordinary people's homes, *e.g.* , two or even four processors machines. As this demand is increasing several solutions are being used to provide computing power for different types of applications. One example of the type of "machine" that is being used are clusters [2]. Although clusters are a good solution for providing computing power for high demanding applications, their use is still very specialized since parallel computation is performed throughout the use of APIs such as Message Passing Interface (MPI) [10]. Using MPI can make access to shared memory quite awkward since shared memory can be spread over several machines far from each other.

Another solution is to use a computer that contains several processors that are managed by a single operating system, *i.e.*, leaving most of the parallel issues to be dealt with by the operating system. One example of such computer are SMP (Symmetric Multi-Processor) computers. However, the number of processors in a SMP computer can be limited since they all access memory using the same memory bus. Such problem can be solved using a different type of computer, *i.e.*, NUMA (Non-Uniform Memory Access) computer. A NUMA computer is a multiprocessor computer organized in nodes in which each node has a set of processors and part of the main memory. This type of processors topology avoids the memory access bottleneck problem.

Although NUMA computers make the construction of scalable multiprocessor computers possible, it brings about several other issues to be solved by the operating system. One of such problems is the scheduling of processes. In these machines, either SMP or NUMA, the process scheduler needs to implement a load balancing mechanism in order to guarantee that processes will be fairly distributed among all processors, minimizing their average execution time.

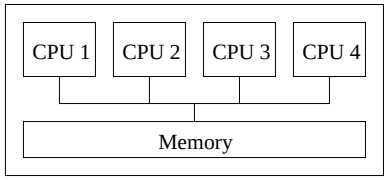


Figure 1: SMP computer

In SMP computers, processes can be moved to any processor because their memory area can be accessed by all processors with the same average cost. This property makes load balancing in these machines much simpler than in NUMA computers. Figure 1 shows a four processors SMP computer. In NUMA computer, due to the main memory is distributed among nodes, different memory areas can be accessed simultaneously. However, memory access time depends on the processor that a process is executing and on the memory area that this process is accessing. Figure 2 shows an eight processors NUMA machine organized in four nodes. In that machine, CPU 1 accesses memory in node 1 faster than memory in nodes 2, 3 or 4.

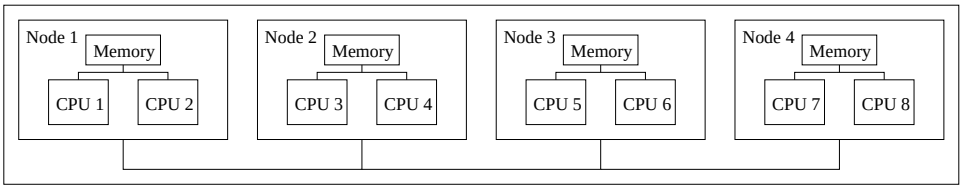


Figure 2: NUMA computer

The computer in Figure 2 has only two memory access levels, *i.e.*, there are two different access times for each processor: (i) access time to memory in the same node of the processor; and, (ii) access time to memory in another node. However, there are NUMA computers with three or more memory access levels. In NUMA computers, a load balancing algorithm must keep processes as close as possible to their memory areas. This feature makes load balancing in these architectures a complex problem, specially when there are many memory access levels.

As there has been an increasing use of NUMA computers, the need for efficient load balancing algorithms in operating systems is being demanded. Regarding Linux, in the last two years developers have put great effort in improving performance and scalability. The result of this effort is a very fast and robust process scheduler that implements an efficient load balancing algorithm. However, many developers and researchers are working to improve these algorithms. One of these efforts is the "NUMA aware scheduler" [4], which keeps information about the home node of processes and attracts them back to their home nodes when they are moved.

The current Linux load balancing algorithm (kernel 2.6.11.5[1]) uses struc-

[1]This is the lastest stable version. Currently there are some works on the kernel version 2.6.12, but to the best of our knowledge the Linux load balancing has not changed.

tures called *sched domains* that are hierarchically organized to represent the computer‘s architecture. Although *sched domains* implementation does not limit the number of levels it can have, the current Linux version builds no more than two levels. Consequently, for some architectures (with more than two memory access levels), the hierarchy will not represent the machine's topology correctly, causing unappropriate load balancing.

In this paper we propose a new strategy to build *sched domains* that represent the actual computer topology. We show, through the use of analytical modeling and benchmarking, that the correct representation of the computer's topology will improve the overall computer performance. In Section 2, we describe the Linux process scheduler, emphasizing its load balancing algorithm. In Section 3, we show the reasons why Linux load balancing is not ideal to NUMA machines with more than two memory access levels. In the same section, we propose a new generic algorithm to build an n-level *sched domains* hierarchy, where n is the number of memory access levels of the computer. Through this multilevel hierarchy, Linux scheduler will be aware of the actual computer's topology and will perform a multilevel load balancing. In Section 4, we present our test results, using benchmarks and analytical models. Finally, in Section 5, we present conclusions of this work.

2 Linux Process Scheduler

Linux is one of the operating systems that implements a scheduler algorithm for multiprocessor machines. Since version 2.5, the Linux scheduler has been called O(1) scheduler because all of its routines execute in an average constant time, no matter how many processors exist [7]. The current version of the Linux scheduler (kernel version 2.6.11) brought many advances for both SMP and NUMA architectures.

The Linux scheduler is preemptive and works with dynamic priority queues. The system calculates process priority according to process CPU utilization rate. I/O-bound processes, which spend most of their time waiting for I/O requests, have higher priority than CPU-bound processes, which spend most of their time running. Since I/O-bound processes are often interactive, they need fast response time, thus having higher priority. CPU-bound processes run less frequently, but for longer periods. Priority is dynamic; it changes according to process behavior. Process timeslice is also dynamic and determined based on process priority. The higher the process priority, the higher the process timeslice.

Although previous versions of Linux had only one process queue for the entire system, the current O(1) scheduler keeps a process queue (called *runqueue*) per processor. Thus, if a process is inserted in a runqueue of a specific processor, it will run only on that processor. This property is called processor affinity. Since the process keeps running in the same processor, the data of this process can be in the cache memory, so the system does not need to retrieve this data from the main memory, clearly an advantage. Since accessing cache memory is faster than accessing main memory, processor affinity improves the overall system performance. Each runqueue contains two priority arrays: active and expired. Priority arrays are data structures composed of a priority bitmap and an array that contains one process queue for each priority. The priority bitmap is used to find the highest priority processes in the runqueue efficiently. It has one bit for each priority level. When at least one process of a given priority exists, the corresponding bit in the bitmap is set to 1. Then, the scheduler selects a new process to run by searching for the first bit equal to 1 in the bitmap, which represents the highest priority of the runqueue, and finding the first process on the queue with that priority. Figure 3 depicts part of this algorithm [7].

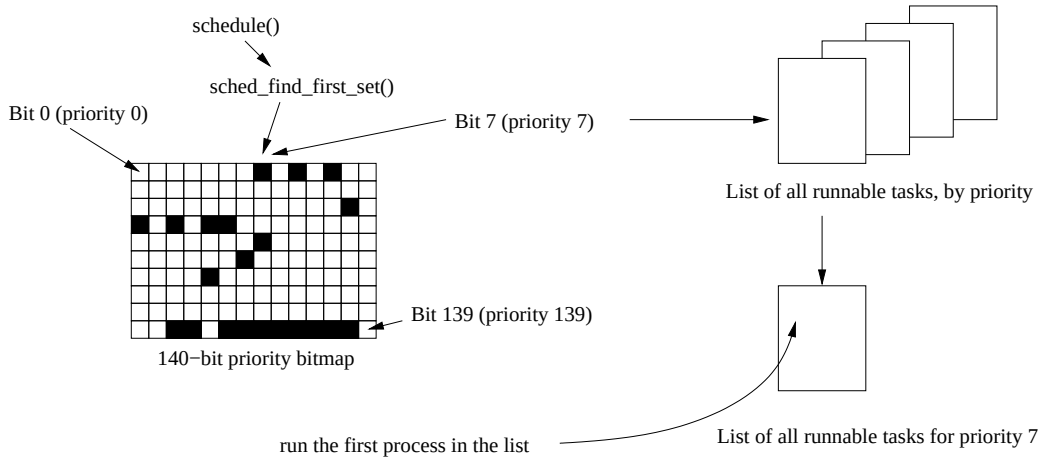


Figure 3: Selecting a new process

Each runqueue has two pointers to the priority arrays. When the active array is empty, the pointers are switched: the expired array becomes the active array and vice-versa. The main advantages of this operation is to avoid moving all processes to the active priority array; executing in constant time; and keeping the scheduling algorithm with O(1) complexity.

When a process finishes its timeslice, its priority and timeslice are recalculated and the process is moved to the expired priority array. This process will run again only when the active array is empty, that is, when all processes of the runqueue have finished their timeslices.

2.1 Linux Load Balancing Algorithm

When a process is created, Linux scheduler inserts it in the same runqueue and with the same priority of its parent. The timeslice of the parent is split equally between the new process and its parent. However, always inserting new processes in the same runqueue can overload a processor, while other processors in the system may be idle or have a smaller number of processes to execute. This is not a desirable scenario because it increases the average execution time of processes. To avoid this, the Linux scheduler implements a load balancing algorithm. This algorithm tries to keep the load of the system fairly distributed among processors. To accomplish this goal, the Linux load balancer migrates processes from an overloaded processor to another with less processes to execute.

In SMP computers, the choice of migrating processes from an overloaded processor to an idle processor does not cause any major side-effect. Since the distance between all processors and memory is the same, migrating a process from any processor to another does not affect the overall performance of the process. This does not happen in NUMA computers; migrating a process from a processor in the same node is better than migrating it from a processor in another node. As described before, this is due to the different memory distances between processors that are in different nodes.

The Linux load balancing algorithm uses a data structure, called *sched domain*, to perform load balancing [1]. Basically, a *sched domain* contains CPU groups that define the scope of load balancing for this domain. The *sched domains* are organized hierarchically, trying to represent the topology of the computer. Figure 4 shows *sched domains* created by Linux to the NUMA computer of Figure 2.

The domains in the lowest level represent nodes of the computer. These domains are called *CPU domains* because processes can be migrated only among CPUs of the same node. Each CPU group in the CPU domains is composed of only one CPU. The higher level domain represents the entire system and is called *node domain* because processes can be moved among nodes. In the node domain, each CPU group represents one node, thus being composed by all CPUs of that node.

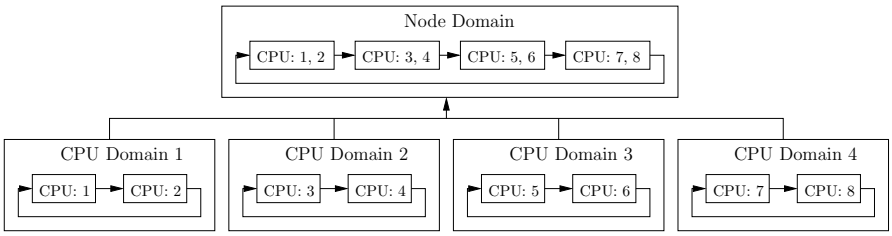


Figure 4: *sched domains* for a NUMA machine

Usually, the Linux O(1) scheduler tries to keep the load of all processors as balanced as possible by reassigning processes to processors in three different situations: (i) when a processor becomes idle; (ii) when a process executes the *exec* or *clone* system calls; and (iii) periodically at specific intervals defined for each *sched domain*.

When a processor becomes idle, the load balancer is invoked to migrate processes from another processor to the idle one. Usually only the CPU domains accept load balancing for this event, in order to keep the processes in the same node and close to the memory allocated to it. When a process executes the *exec* or *clone* system calls, load balancing can be executed for the node domain, because a new memory image will need to be created for the cloned process and it can be allocated on a new node.

Because idle processor events usually trigger load balancing at the CPU level only and an *exec* or *clone* system calls may not be invoked for a long time, periodical load balancing at regular intervals is also performed to prevent imbalances among CPUs on different nodes. In this periodical load balancing, at each rebalance tick, the system checks if the load balancing should be executed in each *sched domain* containing the current processor, starting at the lowest domain level.

Load balancing is performed among processors of a specific *sched domain*. Since a load balancing must be executed on a specific processor, the balancing will be performed in *sched domains* that contain this processor. The first action of the load balancer is to determine the busiest processor of the current domain (all domains are visited, starting at the lowest level) and to verify if it is overloaded with respect to the processor that is executing the load balancing.

The choice of what processes will be migrated is simple. Processes from the expired priority array are preferred and are moved according to three criteria: (i) the process should not be running; (ii) the process should not be *cache-hot*; and (iii) the process should not have processor affinity.

This load balancing algorithm is part of the O(1) scheduler and its goal is to keep the load of all processors as balanced as possible, minimizing the average time of process execution.

3 Multilevel Load Balancing

As shown in Section 2, in NUMA computers, Linux builds a two levels *sched domains* hierarchy for each processor: the first level contains processors of the same node and the second level has all system processors. For NUMA computers that have only two memory access levels, this structure represents exactly the machine's topology. However, there are NUMA computers with more than two memory access levels, *e.g.*, the HP Integrity Superdome servers [5] and the SGI Altix 3000 servers [8]. Figure 5 shows a computer with three memory access levels. This computer is an eight processors NUMA computer composed of four nodes, each one with two processors. In this type of computer, a process initially created in processor 1 or 2 (node 1) could execute, for instance, 25% slower in processors 3 or 4 (node 2), and 50% slower in processors 5, 6 (node 3), 7 or 8 (node 4)[2].

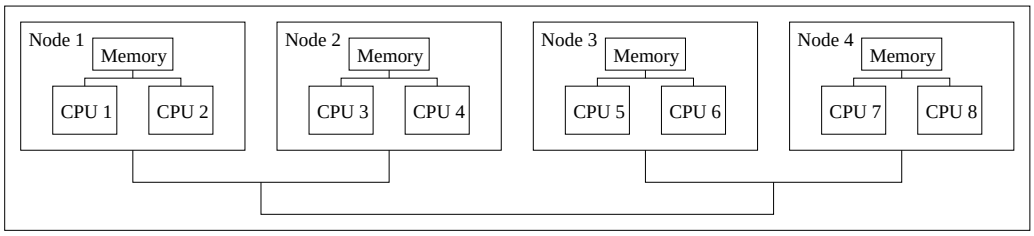


Figure 5: NUMA machine with three memory access levels

Although this computer has three memory access levels, Linux builds the two levels *sched domains* hierarchy shown in Figure 4. Consequently, the migrating probability of a task to any processor outside its original node is the same, no matter the different memory access times.

It is important to point out that for some architectures, the function that builds the *sched domains* hierarchy is overridden, *e.g.*, for the IA64 computers. However, none of these specific implementations builds a *sched domains* hier-

[2]These figures are estimates based on the time processes take to access their memory on different nodes.

archy with more than three levels, independently of how many memory access levels the computer has.

In order to solve this problem, we propose a new algorithm to build the *sched domains* hierarchy. This new algorithm is generic and it builds an n-level *sched domains* hierarchy for a computer with n memory access levels. Figure 6 shows the hierarchy created by our proposed algorithm for the computer in Figure 5.

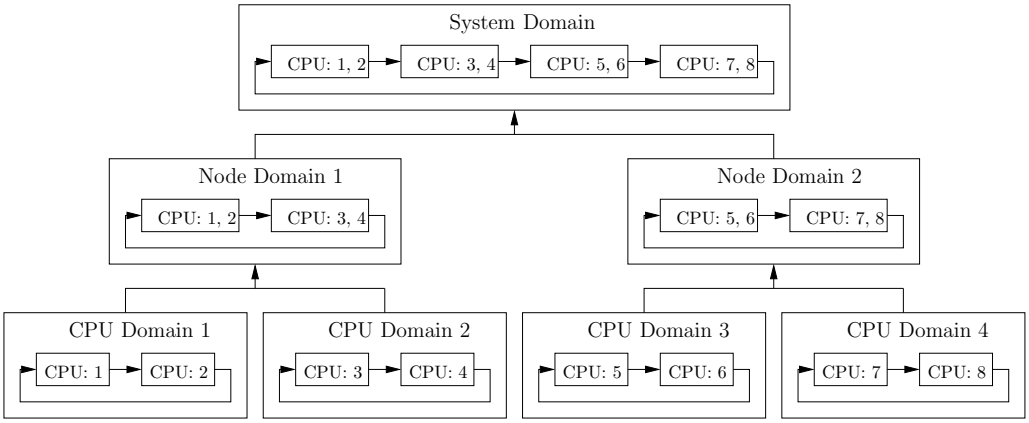


Figure 6: *sched domains* for a NUMA machine with three memory access levels

Using hierarchy shown in Figure 6, when the Linux load balancing algorithm executes in processor 1, for example, it will move tasks from processors in node 1 to processor 1 (first level). After that, if necessary, tasks will be migrated from processors in node 2 to processor 1 (second level), because node 2 is closer to node 1 than node 3 or node 4. Finally, only if there is still an imbalance among processors, Linux will move processes from nodes 3 or 4 to processor 1 (third level). Thus, with the three levels *sched domains* hierarchy, Linux load balancing algorithm can try to keep the processes closer to their memory areas, improving their execution times.

3.1 ACPI SLIT Table

It is necessary that the operating system recognizes all memory access levels to construct a *sched domains* hierarchy that represents the real computer's topology. This information can be read from the ACPI SLIT table [6].

ACPI (Advanced Configuration and Power Interface) is an interface specification that provides information about hardware configuration and allows

operating systems to perform power management for devices [6]. All ACPI data are hierarchically organized in description tables built by the computer firmware. One of these tables, called *System Locality Information Table* (SLIT), describes the relative distance (memory latency) among *localities* or *proximity domains*. Specifically in case of NUMA computers, each node is a locality. Thus, the distance between nodes is available in the ACPI SLIT table.

In a SLIT table, the value of a $P_{i,j}$ position represents the distance from node i to node j. The distance from a node to itself is called *SMP distance* and has the default value of 10. All other distance values are relative to SMP distances. As processors and memory blocks are inside nodes, ACPI SLIT table provides the distance between processors and memory areas. That is, different distance values in the SLIT table represent different memory access levels of the system.

Table 1 shows a possible SLIT table for the NUMA computer in Figure 5. According to this table, the distance from node 1 to node 2 is twice the SMP distance. This means that a processor in node 1 acesses a memory area in node 2 twice slower than a memory area in node 1 [6]. Analogously, this same processor accesses a memory area in nodes 3 or 4 four times slower than a memory area in node 1.

	Node 1	Node 2	Node 3	Node 4
Node 1	10	20	40	40
Node 2	20	10	40	40
Node 3	40	40	10	20
Node 4	40	40	20	10

Table 1: SLIT table to a multilevel NUMA machine

Note that the three different distance values in the SLIT table represent exactly the three memory access levels shown in Figure 5. Thus, it is possible for the operating system to find how many memory access levels exist in the machine and what nodes are closer through the SLIT table data.

3.2 Proposed Algorithm Description

In order to build a *sched domains* hierarchy that represents correctly the computer's topology, we propose a generic algorithm that uses the SLIT table data to recognize all memory access levels of the computer. In our approach, a multilevel *sched domains* hierarchy is constructed as follow:

1. For each node N:

   (a) Choose a processor P in node N.

   (b) For each SLIT table distance d from node N to all other nodes (in *increasing* order):

      i. Create a new *sched domain* for processor P. If this is not the first domain of this processor, it is the parent of the last created domain.

      ii. Create a list of CPU groups for the *sched domain* created in the previous step. If $d = 10$ (distance from node N to itself), this list will have one CPU group for each processor in node N. Otherwise, the list must be composed by one CPU group for each node that has the distance to node N less than or equal to d.

   (c) Replicate the *sched domains* hierarchy created for processor P to all other processors in node N.

Although the *sched domains* hierarchy is equal to processors in the same node, it is not possible to have a shared hierarchy. Hierarchy replication is necessary for each processor because the *sched domain* structure maintains load balancing information on each specific processor, *e.g.*, the last time the load balancing algorithm was executed in that processor.

According to the algorithm described above, if a machine has n memory access levels, Linux will build an n-level *sched domains* hierarchy. The main advantage of a multilevel *sched domains* hierarchy is that the migrating probability of processes to distant nodes decreases, thus improving their average execution time.

3.3 Case of Study

In order to demonstrate an application for the proposed algorithm, we will use as example an HP Integrity Superdome computer [5]. This machine is a sixty four processors NUMA computer with sixteen nodes and three memory access levels, as shown in Figure 7.

As shown in the following SLIT table (Table 2), this computer has 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.

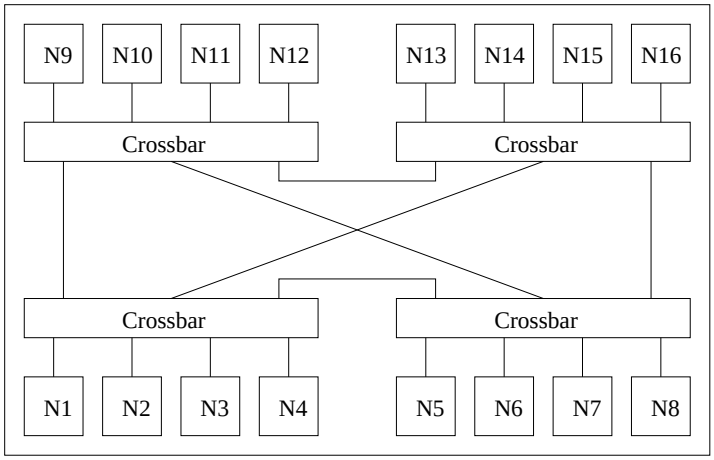


Figure 7: HP Integrity Superdome

	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10	N11	N12	N13	N14	N15	N16
N1	10	17	17	17	29	29	29	29	29	29	29	29	29	29	29	29
N2	17	10	17	17	29	29	29	29	29	29	29	29	29	29	29	29
N3	17	17	10	17	29	29	29	29	29	29	29	29	29	29	29	29
N4	17	17	17	10	29	29	29	29	29	29	29	29	29	29	29	29
N5	29	29	29	29	10	17	17	17	29	29	29	29	29	29	29	29
N6	29	29	29	29	17	10	17	17	29	29	29	29	29	29	29	29
N7	29	29	29	29	17	17	10	17	29	29	29	29	29	29	29	29
N8	29	29	29	29	17	17	17	10	29	29	29	29	29	29	29	29
N9	29	29	29	29	29	29	29	29	10	17	17	17	29	29	29	29
N10	29	29	29	29	29	29	29	29	17	10	17	17	29	29	29	29
N11	29	29	29	29	29	29	29	29	17	17	10	17	29	29	29	29
N12	29	29	29	29	29	29	29	29	17	17	17	10	29	29	29	29
N13	29	29	29	29	29	29	29	29	29	29	29	29	10	17	17	17
N14	29	29	29	29	29	29	29	29	29	29	29	29	17	10	17	17
N15	29	29	29	29	29	29	29	29	29	29	29	29	17	17	10	17
N16	29	29	29	29	29	29	29	29	29	29	29	29	17	17	17	10

Table 2: HP Integrity Superdome SLIT Table

According to our algorithm, Linux has to perform the following steps to create the *sched domains* hierarchy:

1. For node $N = N1$:
   (a) $P = 1$, the first processor in node $N1$.
   (b) For $d = 10$:
      i. Create the first *sched domain* for processor 1.
      ii. The CPU groups list for the created *sched domain* has one CPU group for each processor in node $N1$.
      Figure 8 shows the *sched domains* hierarchy created for processor 1 up to now.

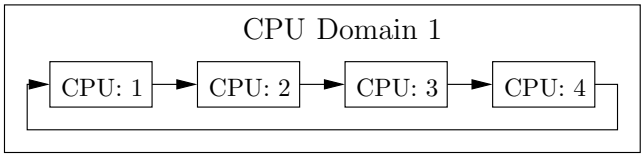


Figure 8: *sched domains* hierarchy for processor 1 (Part 1)

(c) For $d = 17$:

i. Create another *sched domain* for processor 1, which will be the parent of the last created *sched domain*.

ii. The CPU groups list for this new *sched domain* is composed of four groups. Each group contains processors of a specific node. The four nodes are N1, N2, N3 and N4 since the distance between node N1 and these nodes are less than or equal to 17. The current *sched domains* hierarchy of processor 1 is shown in Figure 9.

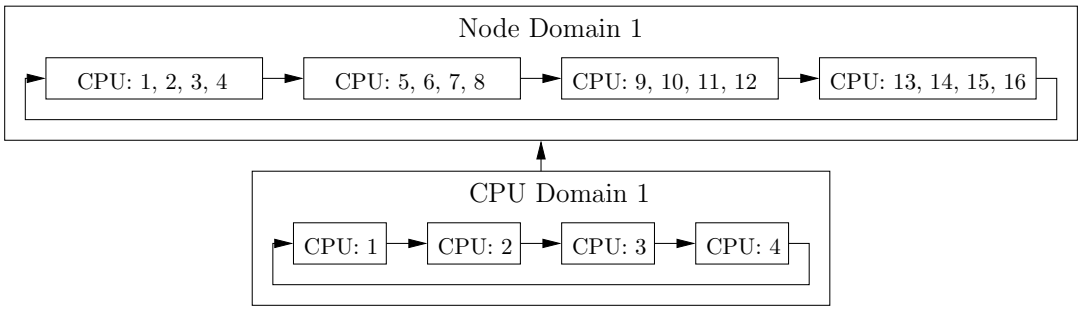


Figure 9: *sched domains* hierarchy for processor 1 (Part 2)

(d) For $d = 29$:

i. Create the last *sched domain* for processor 1, which will be the parent of the last created *sched domain*.

ii. The CPU groups list for this *sched domain* has sixteen groups, one for each node of the system. All nodes will be inserted in this list because 29 is the biggest distance value in the SLIT table.
The final *sched domain* built for processor 1 is shown in Figure 10.

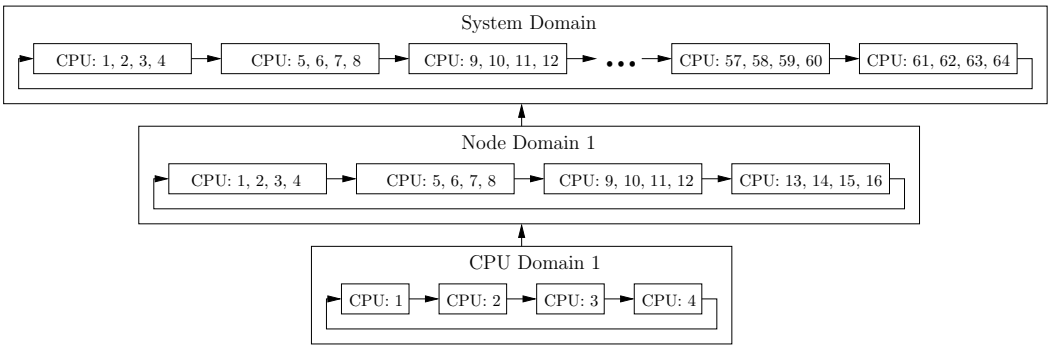


Figure 10: *sched domains* hierarchy for processor 1

(e) Replicate the *sched domains* hierarchy that was created for processor 1 to the other processors in node $N1$, *i.e.*, processors 2, 3, and 4.

The same algorithm will be executed for all other nodes in the system. For each processor in a node N a *sched domains* hierarchy will be created with the same characteristics: a three levels hierarchy with the lowest level composed of processors in node N, the second level composed of nodes with distances to node N less than or equal to 17 and the highest level with all processors of the system. For this same machine, the *sched domains* hierarchy built by the Linux original code for processors in node $N1$ is shown in Figure 11.

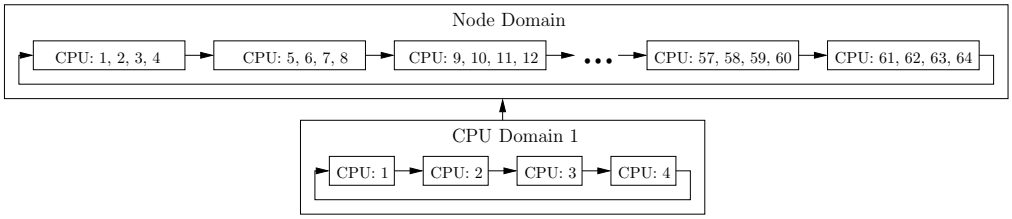


Figure 11: Two levels *sched domains* hierarchy for node $N1$

In order to compare the Linux load balancing execution in the HP Integrity Superdome when using the hierarchy created by the Linux original code and that created by our proposal, consider, for example, load balancing executing in processor 1 (node $N1$). Regarding the *sched domains* hierarchy in Figure 11, load balancing will initially be executed among processors in the same node (lowest level in the hierarchy). According to the load balancing algorithm

described in Section 2.1, it will search for the most overloaded processor in node $N1$ and this processor's load will be balanced with processor 1. After that, load balancing will be executed in the second level of the hierarchy, which represents the entire system. The Linux algorithm will search for the most overloaded processor of the most overloaded node and balance its load with processor 1. At this point, although processor 5 is closer to processor 1 than processor 17, if processor 17 is a bit more overloaded than processor 5, then the Linux algorithm will choose processor 17 to balance with processor 1. Consequently, tasks will be migrated from processor 17 to processor 1. This situation happens because Linux is using a two levels *sched domains* hierarchy for a machine with three memory access levels, *i.e.*, the Linux load balancing algorithm is not aware of the three existing memory access levels.

On the other hand, the three levels *sched domains* hierarchy in Figure 10 can be used by the load balancing algorithm to avoid this problem. After executing load balancing among processors in node $N1$ (first level), the load balancing algorithm will search for an overloaded processor in the second level. This level is composed of nodes $N1$, $N2$, $N3$ and $N4$, which are closer to processor 1 than the other nodes. Although processor 17 is more overloaded than processor 5, in our example, it is in node $N5$, which is not considered in this level. Thus, tasks will be initially migrated from processor 5 to processor 1. Finally, in the third level, the algorithm will execute considering the entire system.

The second level created by our algorithm in the *sched domains* hierarchy will guarantee that tasks will be as close to their memory areas as possible, improving their execution time.

4 Evaluation

Performance evaluation by benchmarking is one of the main approaches for measuring performance of computer systems. Although it can be a very convincing way of measuring an actual system, benchmarking and other monitoring techniques are often inflexible as an analysis tool. There are situations where it would be too expensive to acquire or to modify the system in order to evaluate it by benchmarking. Therefore, producing theoretical models might be a good solution to solve this problem.

In order to evaluate our algorithm, we have chosen two different approaches: the use of benchmarks and analytical models. Benchmarks were used to evaluate a twelve processor NUMA computer with two memory access levels. Our

aim was to check whether the new algorithm would not affect the performance of the system in comparison to the current Linux algorithm. For machines with three and four memory access levels, we have constructed an analytical model to verify some performance indices.

In respect to benchmarking, we used the *kernbench* benchmark [3]. *Kernbench* is a CPU throughput benchmark that compiles a kernel with various numbers of concurrent jobs and provides useful statistics for the average of each group of compilings.

Regarding the analytical model, we use the *Stochastic Automata Networks* (SAN) [9] formalism to describe performance indices of the current way Linux builds the *sched domains* hierarchy and the proposed on this paper. The SAN formalism is quite attractive when modeling systems with several parallel activities. In this case, processors can be modeled as automata which work independently but can interact in some situations, *e.g.*, when it is necessary to migrate processes from one processor to another. The main idea of our approach is to model the behavior of only one process in the Linux system, but considering the influence of other processes. Therefore, our model consists of $P^{(i)}$ processors and one process.

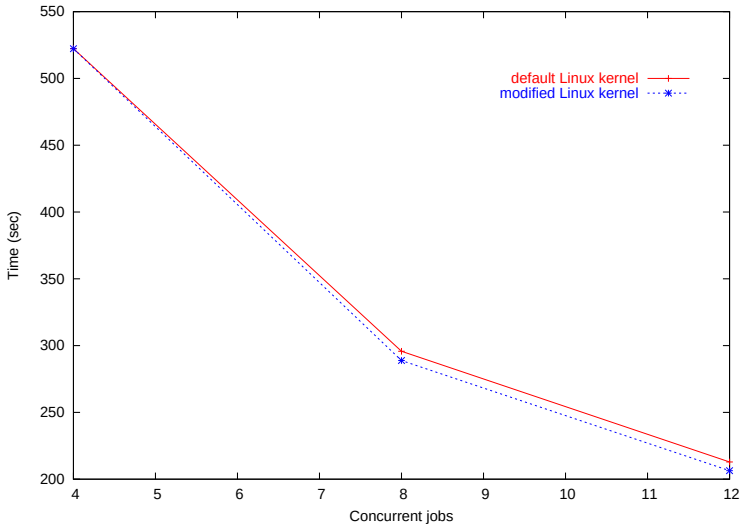
4.1 Benchmark Results

Since we executed the benchmark in a machine with two memory access levels, the *sched domains* hierarchy created by Linux original code and by our algorithm are equal. Figure 12 shows the results of *kernbench* executing 4, 8 and 12 concurrent jobs, in a Linux original kernel and the Linux modified kernel. In the modified kernel, the Linux function that builds the *sched domains* hierarchy was replaced by our algorithm. As we expected, the performance of the system was not affected, since the created *sched domains* hierarchy is the same.

4.2 Analytical Results

We have applied the analytical model to different models of NUMA computers. For each computer we verified the normal end probability of a process in two situations: when Linux builds the *sched domains* hierarchy according to the current Linux algorithm and when Linux builds the *sched domains* hierarchy according to our proposed approach.

The first computer is a NUMA computer with four nodes and three memory access levels (see Figure 13).



kernel	4 concurrent jobs	8 concurrent jobs	12 concurrent jobs
default	522.335 s	295.734 s	212.87 s
modified	522.210 s	288.864 s	206.336 s

Figure 12: *kernbench* results

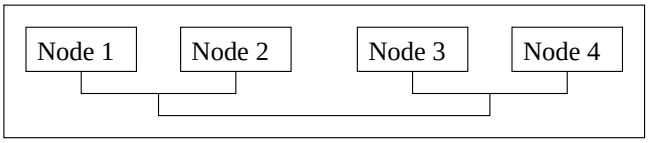


Figure 13: NUMA computer with three memory access levels

Figure 14 shows the normal end probability of a process when Linux recognizes only two memory access levels (*3 memory access levels - 2 Linux sched domain levels*) and when Linux recognizes the actual computer's topology (*3 memory access levels - 3 Linux sched domain levels*). Note that the proposed approach presents a better performance depending on the time the process takes to execute (around 0.8% for a thirty seconds process). Such phenomenon occurs due to the fact that the longer the process takes to execute, the greater the chance for the process to migrate from one node to another. When migration takes place, the current Linux algorithm does not consider the different distances among different nodes. Therefore, the process could take more time to finish when moved to a more distant node.

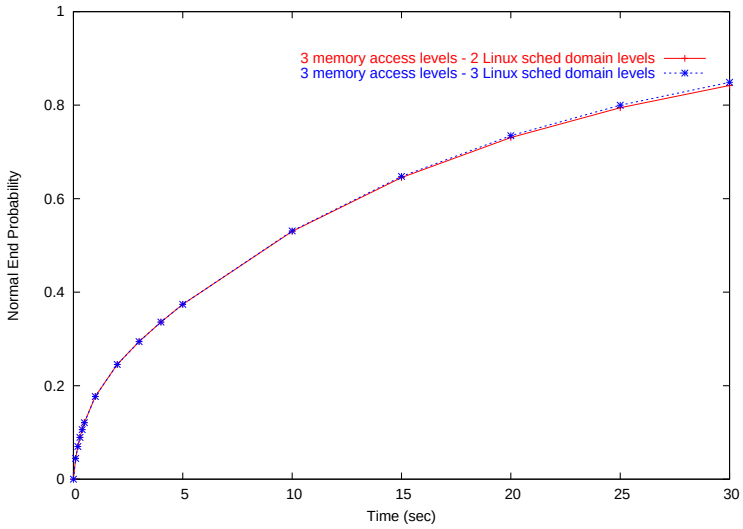


Figure 14: Two and three hierarchy levels behavior

The second computer is a NUMA computer with the same number of nodes (four nodes) but with four memory access levels (see Figure 15).

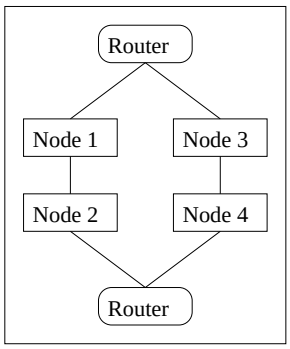


Figure 15: NUMA computer with four memory access levels

Figure 16 shows the normal end probability of a process 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 levels*). Comparing to the first example (Figure 14), the performance in this case is even better (around 1%). As there is one more memory access level, there is a higher chance for the pro-

cess to be migrated to the more distant node in the current Linux algorithm than in our approach.

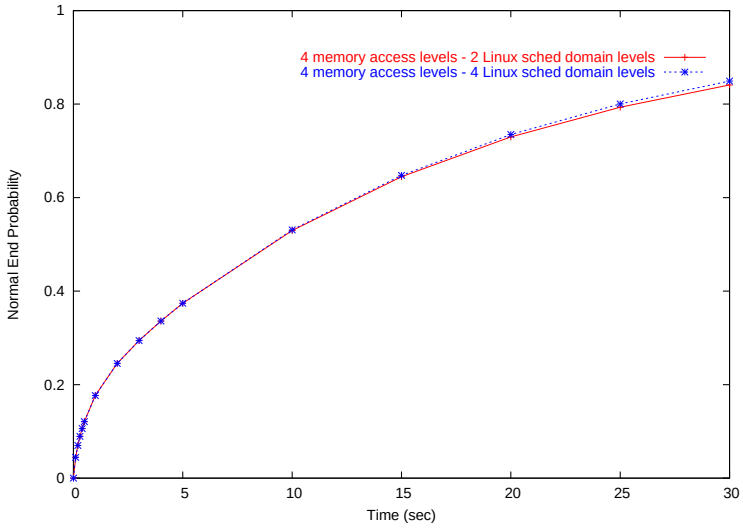


Figure 16: two and four hierarchy levels behavior

Although it might seems that the improvement is not significant, it is important to notice that we are analysing the behavior of only one process in the system. If each process presents a small improvement, the overall performance will increase. Besides, as the number of memory access levels increases, the difference between our approach and the current Linux algorithm increases. Another important point we could observe in our analytical analysis is that the improvement increases as the time the process takes to finish its execution increases. We modeled a process that would take only thirty seconds to execute, and clearly our results showed that the difference between our approach and the current Linux approach would increase as the time the process takes to finish increases.

5 Conclusion

This paper has presented a new approach to build load balancing hierarchy in order to consider the correct NUMA computer topology. Our approach has been implemented and analysed throughout the use of a benchmark and analytical modeling. The benchmark results applied to the implementation

that was run on a two memory access levels HP Superdome computer has shown that our approach has not introduced any overhead to the load balancing algorithm being executed in the current Linux system. We did not have access to a NUMA computer with more than two memory access levels, therefore we used analytical modeling to verify our approach on computers with more than two memory access levels. The analytical results have shown that our approach is slightly better than the current approach for small computers, but as the number of memory access levels increases the performance of our approach will also increase. We shown also that the longer a process takes to finish, the better our approach will perform.

One consequence of building a more complex *sched domains* hierarchy is the increase in load balancing execution time. Thus, it is possible that for some number of memory access levels the complexity of the created hierarchy makes load balancing execution time greater than the gains in the processes execution time. To verify if this limit exists and to what extent, we intend to test our algorithm in other architectures, with different numbers of memory access levels, using benchmarks and simulations. We also provided a kernel patch to allow our proposal to be tested in different environments.

References

[1] M. J. Bligh, M. Dobson, D. Hart, and G. Huizenga. Linux on NUMA Systems. In *Proceedings of the Linux Symposium*, volume 1, pages 89–102, Ottawa, Canada, July 2004.

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

[3] C. Kolivas. Kernbench benckmark. Available at URL http://ck.kolivas.org/kernbench, 2004.

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

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

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

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

[8] 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.

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

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