Operating System Multilevel Load Balancing

M. Corrêa, A. Zorzo
Faculty of Informatics - PUCRS
Porto Alegre, Brazil
{mcorrea, zorzo}@inf.pucrs.br

R. Scheer
HP Brazil R&D
Porto Alegre, Brazil
roque.scheer@hp.com

ABSTRACT

This paper describes an algorithm that allows Linux to perform multilevel load balancing in NUMA computers. The Linux scheduler implements a load balancing algorithm that uses structures called *sched domains* to build a hierarchy that represents the machine's topology. Although *sched domains* implementation allows Linux to build a multilevel hierarchy to represent multilevel machines, the generic code of the current kernel version builds no more than two levels in the *sched domains* hierarchy. Thus, for NUMA systems with three or more memory access levels, the constructed hierarchy does not represent correctly the machine's topology. When Linux load balancing algorithm uses an incorrect *sched domains* hierarchy, process execution time can increase, because processes can be moved to nodes that are distant from their memory areas. In order to solve this problem, we have implemented an algorithm to build multilevel *sched domains* hierarchies for NUMA computers. Our proposed algorithm uses ACPI SLIT table data to recognize how many memory access levels a machine contains. Then, it builds an n-level *sched domains* hierarchy, where n is the number of memory access levels. Through benchmarking and simulation results we demonstrate that the Linux load balancing performance when the *sched domains* hierarchy is built using our proposed algorithm is better than using the current Linux algorithm.

Categories and Subject Descriptors

D.4.0 [**Operating Systems**]: General—*Linux*; D.4.1 [**Operating Systems**]: Process Management—*Multiprocessing/multiprogramming, Scheduling*; D.4.8 [**Operating Systems**]: Performance—*Measurements, Simulation*

General Terms

Load Balancing, Scalability, Performance

Keywords

NUMA computers, Load Balancing, Performance, Linux

1. INTRODUCTION

The demand for computational power has been increasing throughout the past years. Several solutions are being used in order to respond to such demand, *e.g.*, clusters of workstations [5] 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*) [17]. 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 [11]. In UMA computers each processor can access any memory area with the same average cost, which simplifies the load balancing. The major drawback of UMA architectures is that the number of processors is limited by the contention on access to the shared 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. In this sense, there are many proposals for different platforms, for example clusters [4, 2, 20] and computational grids [8, 18]. Some authors have also presented solutions or studies for the load balancing problem on NUMA computers. Zhu [19], for instance, proposes a cluster queue structure for processes based on a hierarchical structure. Focht [7], 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 [6].

This paper proposes an algorithm that allows Linux to perform multilevel load balancing in NUMA computers. The current Linux load balancing algorithm uses a 2-level *sched domains* hierarchy to represent the machine's topology and perform load balancing. However, there are NUMA computers with more than two memory access levels. For these architectures, a 2-level *sched domains* hierarchy does not represent their topology correctly, causing unappropriate load balancing. To cope with this problem, we propose 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 also 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: *benchmarking* and *simulation*. The simulation model is developed using the *JavaSim* simulation tool [12], and KernBench [13] is used for benchmarking.

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 shows the results of simulation and benchmarking 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 4 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 [11]. 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 [9] 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.

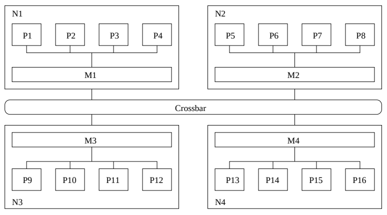

Figure 1: HP Integrity Superdome with two memory access levels

However, some NUMA architectures, usually with a greater number of processors, have more than two memory access levels, *e.g.*, the HP Integrity Superdome server in Figure 2 and the SGI Altix 3000 servers [15]. 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.

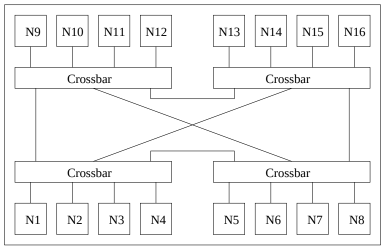

Figure 2: HP Integrity Superdome with three memory access levels

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. This information can be read from the ACPI SLIT table [10].

2.1 ACPI SLIT Table

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 [10]. 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.

Figure 3 shows a possible SLIT table for the NUMA computer in Figure 1. According to this table, the distance from node 1 to node 2 is 1.7 times the SMP distance. This means that a processor in node 1 accesses a memory area in node 2 seventy percent (70%) slower than a memory area in node 1 [10].

	N1	N2	N3	N4
N1	10	17	17	17
N2	17	10	17	17
N3	17	17	10	17
N4	17	17	17	10

Figure 3: HP Integrity Superdome SLIT Table

Note that the two different distance values in the SLIT table represent exactly the two memory access levels shown in Figure 1. 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.

2.2 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 is not scalable: as the number of processors increases, the process queue becomes a bottleneck. The current Linux scheduler has one process queue for each processor in the system, solving this scalability problem. However, due to the multiple process queues, Linux had to implement a load balancing algorithm [14].

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, 3]. 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 4.

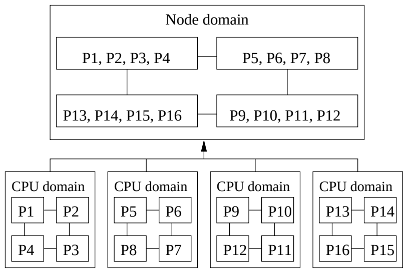

Figure 4: 2-level *sched domains* hierarchy

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.

2.3 Multilevel Load Balancing Algorithm

The load balancing algorithm described in Section 2.2 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, since 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 propose a new algorithm to build the *sched domains* hierarchy (Figure 6). 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. Hence, for the machine in Figure 2, which has three memory access levels, our proposed algorithm builds the *sched domains* hierarchy shown in Figure 5.

```
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*
           built 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 $N$.
```

Figure 6: Algorithm to build the *sched domains* hierarchy

Using the hierarchy from Figure 5, 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.

2.4 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. In order to implement the proposed algorithm for the construction of a multilevel *sched domains* hierarchy, we have changed this specific function and created some auxiliary functions.

We have developed a patch for the current Linux kernel version (2.6.14). This patch is available in the PeSO project web site [16].

3. NUMERICAL RESULTS

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 use the *JavaSim* simulation tool [12], and the KernBench benchmark [13]. *Kernbench* is a CPU throughput benchmark that compiles a kernel with various numbers of concurrent jobs and provides the average execution time for each group of compilings.

Figure 7 shows the results from the KernBench benchmark. The computer we have run the benchmark is the HP Integrity Superdome shown in Figure 1. We have built

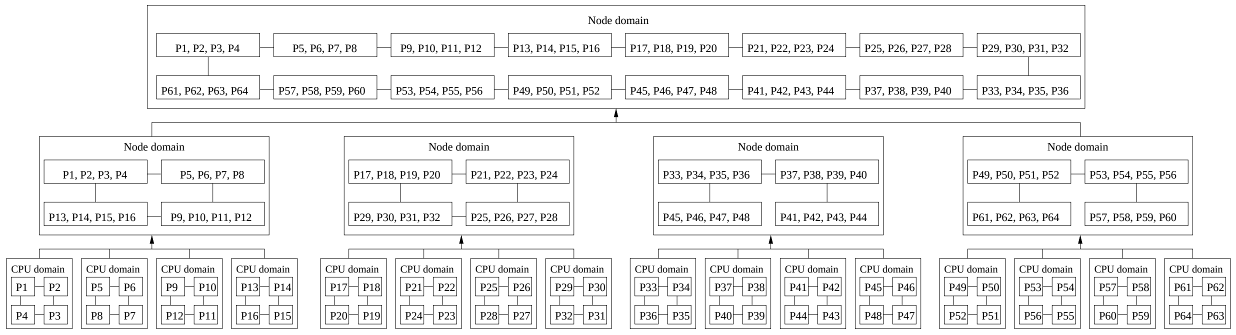

Figure 5: 3-level *sched domains* hierarchy

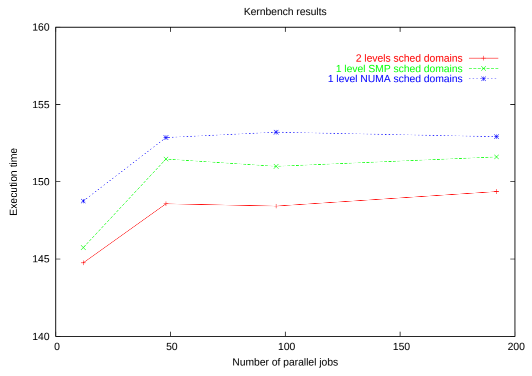

Figure 7: KernBench results

the *sched domains* to consider: i) two memory access levels (line “2 levels”); ii) only one memory access level (line “1 level SMP”); iii) only one memory access level (line “1 level NUMA”), but to execute load balancing only at the node level and not at the CPU level. This strategy was used to verify the influence of using a system that does not consider the right number of memory access levels when executing the load balancing. As can be seen in Figure 7, when all memory access levels are taken into consideration by the load balancing algorithm (line “2 levels”), the system has a better performance, since it will consider first to balance the system load among processors in the same node, and only after that, it will consider load balancing among processors from different nodes. Furthermore, the execution of load balancing only at node level (line “1 level NUMA”) shows a worse performance than executing load balancing considering all processors in the system (line “1 level SMP”). This is due to the fact that the load balancing algorithm will consider only processors from different nodes to balance the system load.

Regarding the simulation results, we considered the following[1]: static priority (*nice value*) equal to 0; average process execution time of 500 milliseconds; average processing time, before yielding the processor, of $timeslice+nice\_value$ milliseconds; and, average IO 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, with standard deviation between 0,4% and 1,3%, and confidence coefficient of 99%.

[1]Values are based on benchmark results or Linux constants.

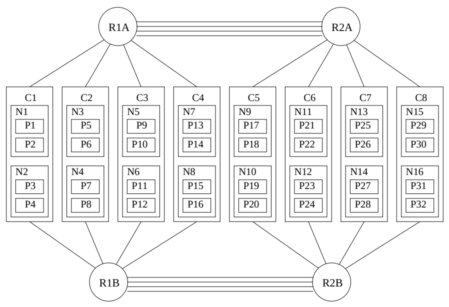

Figure 8: SGI Altix 3000 server

We have considered two commercial machines in the simulations to verify the system performance when our proposed algorithm is used. The computers we used were the SGI Altix 3000 server with six memory access levels (Figure 8) and the HP Integrity Superdome computer with three memory access levels (Figure 2).

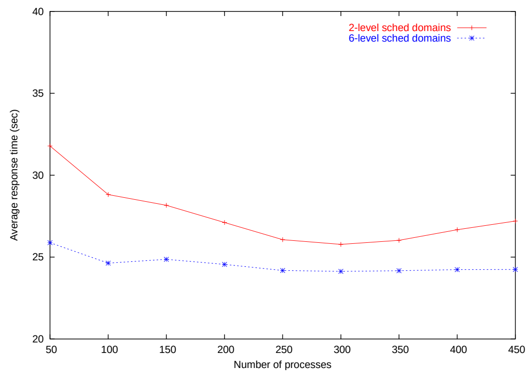

Figure 9: SGI Altix 3000 simulation results

The SGI Altix computer has 16 nodes and only two processors per node. The simulation results for this machine are presented in Figure 9. For this machine, the average performance improvement was 10%. The HP Superdome computer in Figure 2 has three memory access levels and four processors per node. For this machine, the simulation achieves an average performance improvement of 2.2% (Figure 10).

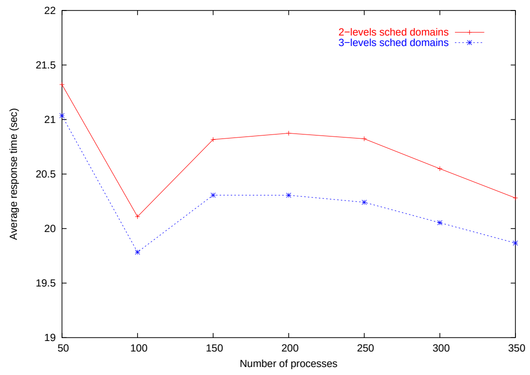

Figure 10: HP Superdome simulation results

4. CONCLUSION

This paper has discussed the performance benefits of a new approach for operating systems load balancing in NUMA computers. In order to evaluate this new approach, we used benchmarking and simulation. The simulation results we obtained 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 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 benchmark results we obtained are very promising.

The major contributions of this paper are related to the use of the ACPI SLIT table information to build the *sched domains* hierarchy; benchmark and simulation results for operating systems load balancing algorithm in NUMA computers; 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, few processors per node and different process priorities. Such problem is due to the fact that Linux does not migrate process memory when the process is migrated. We have already constructed a simulation model showing the benefits of memory page migration in NUMA computers. Our next step is to integrate both Linux load balancing and memory migration.

5. ACKNOWLEDGMENTS

This work was developed in collaboration with Hewlett-Packard R&D (PeSO Project). Avelino F. Zorzo is partially funded by CNPq Brazil (grant 307137/2004-0). We thank Katia Saikoski, Rafael Chanin, Afonso Salles and the reviewers for their fruitful comments.

6. REFERENCES

[1] J. Aas. Understanding the Linux 2.6.8.1 CPU Scheduler. 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] M. J. Bligh, M. Dobson, D. Hart, and G. Huizenga. Linux on NUMA Systems. In *Proceedings of the Linux Symposium*, pages 89–102, Ottawa, Canada, 2004.

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

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

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

[7] Eric Focht. NUMA aware scheduler. http://home.arcor.de/efocht/sched, 2002.

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

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

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

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

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

[13] C. Kolivas. Kernbench benckmark. http://ck.kolivas.org/kernbench, 2004.

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

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

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

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

[18] 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.

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

[20] 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.