FACULDADE DE INFORMÁTICA
PUCRS - Brazil

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

Analytical Modeling for Operating System Schedulers on NUMA Systems

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

Technical Report Series

Number 046
April, 2005

Contact:
chanin@inf.pucrs.br
www.inf.pucrs.br/~chanin

mcorrea@inf.pucrs.br

paulof@inf.pucrs.br
www.inf.pucrs.br/~paulof

asales@inf.pucrs.br
www.inf.pucrs.br/~asales

roque.scheer@hp.com

zorzo@inf.pucrs.br
www.inf.pucrs.br/~zorzo

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

1 Introduction

Performance evaluation by benchmarking is one of the main approaches for measuring performance of a computer system. The main idea of benchmarking involves the running of a set of computer programs to measure the performance of a machine. There have been several benchmarks developed to measure different features of a computer system. Usually a benchmark can measure both machine and system characteristics. Examples of benchmarks are AIM Multiuser Benchmark - Suite VII [1], LMBench [16], SPEC Benchmark Suite [22], NAS Parallel Benchmarks [2] and LINPACK Benchmark [8]. The choice of benchmark depends on the features that someone wants to evaluate on a system or on a machine.

Although benchmarks can be a very convincing way of measuring an actual system, benchmarking and other monitoring techniques are often too inflexible as analysis tools. In several situations it is important to modify a system configuration and check whether system behavior changes. The actual reconfiguration could be very difficult and most of the time the obtained results do not clearly show an advantage to justify all spent effort.

One solution to this problem is to produce a (theoretical) model of the system under evaluation and analyze possible configurations. The use of simple models describing small parts of the system under evaluation is frequently used by hard Markovian modelers [23, 13]. Anther valid option is the use of high level formalisms, such as Queueing Networks [10, 11], which can provide insights about the performance, but sometimes it assumes too unrealistic behaviors, *e.g.*, unlimited queues. Another possible solution is the use of structured formalisms [20, 7, 9, 12] to describe parts of a system and then composing these parts to have the full system model. Furthermore, with an analytical model it is possible to verify other types of indices, typically performability indices [17], *i.e.*, indices related to the way the performance of a system is affected in the presence of faults.

Performance and reliability indices can be produced by different models and tools. In this paper, we use the *Stochastic Automata Networks* (SAN) [19] formalism to describe performance and reliability indices of some of the Linux operating system algorithms. However, any other formalism, *e.g.*, *Continuous Time Markov Chain* (CTMC) [21], *Stochastic Activity Networks* (SAN) [20], *Process Algebra* [12] and *Stochastic Petri Nets* (SPN) [7] could be employed.

The SAN formalism is usually quite attractive when modeling systems with several parallel activities. It is also important to notice that SAN provides efficient numeric algorithms to compute stationary and transient measures

[9, 3], taking advantages of the structured and modular definitions. In such way, the SAN formalism allows the solution of considerably large models, *i.e.*, models with more than a few million states.

This paper shows how to model parts of the Linux operating system for NUMA (*Non-Uniform Memory Access*) machines. We present a SAN model from which performance and reliability indices of some parts of the Linux scheduling algorithm can be extracted. Since a model considering all possible processes and processors from a NUMA machine would be too large, we generalize the behavior of all processes modeling the behavior of a single process in an multiprocessor machine. This generic model considers the possibility of faulty processors, process migration and process scheduling in the operating system. Furthermore, such generic model is simplified according to the performance and reliability indices desired.

The rest of this paper is organized as follows. In Section 2, we briefly describe the Stochastic Automata Networks formalism. Section 3 describes the behavior of the Linux scheduling and load balancing algorithms for NUMA machines. In Section 4, we show the proposed SAN model for those algorithms and, in Section 5, we present the numerical results of the proposed model. Finally, Section 6 assesses future work and emphasizes this paper main contributions.

2 Stochastic Automata Networks

The SAN formalism was proposed by Plateau [18] and its basic idea is to represent a whole system by a collection of subsystems with an independent behavior (*local transitions*) and occasional interdependencies (*functional rates* and *synchronizing events*). The framework proposed by Plateau defines a modular way to describe continuous and discrete-time Markovian models [19]. However, only continuous-time SAN will be considered in this paper, although discrete-time SAN can also be employed without any loss of generality.

The SAN formalism describes a complete system as a collection of subsystems that interact with each other. Each subsystem is described as a stochastic automaton, *i.e.*, an automaton in which the transitions are labeled with probabilistic and timing information. Hence, one can build a continuous-time stochastic process related to SAN, *i.e.*, the SAN formalism has exactly the same application scope as Markov Chain (MC) formalism [21, 6]. The state of a SAN model, called *global state*, is defined by the cartesian product of the *local states* of all automata.

There are two types of events that change the global state of a model: *local events* and *synchronizing events*. Local events change the SAN global state passing from a global state to another that differs only by one local state. On the other hand, synchronizing events can change simultaneously more than one local state, *i.e.*, two or more automata can change their local states simultaneously. In other words, the occurrence of a synchronizing event forces all concerned automata to fire a transition corresponding to this event. Actually, local events can be viewed as a particular case of synchronizing events that concerns only one automaton.

Each event is represented by an *identifier* and a *rate* of occurrence, which describes how often a given event will occur. Each transition may be fired as result of the occurrence of any number of events. In general, non-determinism among possible different events is dealt according to Markovian behavior, *i.e.*, any of the events may occur and their occurrence rates define how often each one of them will occur. However, from a given local state, if the occurrence of a given event can lead to more than one state, then an additional *routing probability* must be informed. The absence of routing probability is tolerated if only one transition can be fired by an event from a given local state.

The other possibility of interaction among automata is the use of functional rates. Any event occurrence rate may be expressed by a constant value (a positive real number) or a function of the state of other automata. In opposition to synchronizing events, functional rates are one-way interaction among automata, since it affects only the automaton where it appears.

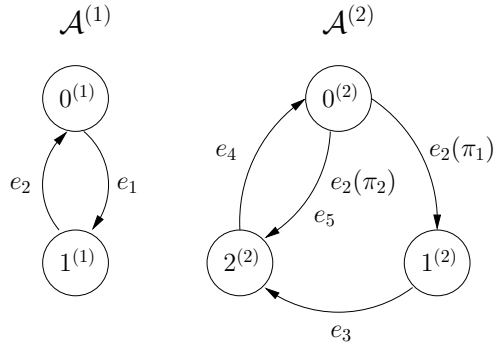
Figure 1 presents a SAN model with two automata, four local events, one synchronizing event, and one functional rate. In the context of this paper, we will use roman letters to identify the name of events and functions, and greek letters to describe constant values of rates and probabilities.

In the model of Figure 1, the rate of the event e_1 is not a constant rate, but a functional rate f described by the SAN notation[1] employed by the PEPS tools [4]. The functional rate f is defined as:

$$f = \begin{cases} \lambda & \text{if automaton } \mathcal{A}^{(2)} \text{ is in the state } 0^{(2)} \\ 0 & \text{if automaton } \mathcal{A}^{(2)} \text{ is in the state } 1^{(2)} \\ \gamma & \text{if automaton } \mathcal{A}^{(2)} \text{ is in the state } 2^{(2)} \end{cases}$$

The firing of the transition from $0^{(1)}$ to $1^{(1)}$ state occurs with rate λ if automaton $\mathcal{A}^{(2)}$ is in state $0^{(2)}$, or γ if automaton $\mathcal{A}^{(2)}$ is in state $2^{(2)}$. If

[1]The interpretation of a function can be viewed as the evaluation of an expression of non-typed programming languages, *e.g.*, C language. Each comparison is evaluated to value 1 (*true*) and value 0 (*false*).



Type	Event	Rate
loc	e_1	f
syn	e_2	μ
loc	e_3	σ
loc	e_4	δ
loc	e_5	τ

$$f = \left[\left(st\ \mathcal{A}^{(2)} == 0^{(2)}\right) * \lambda\right] + \left[\left(st\ \mathcal{A}^{(2)} == 2^{(2)}\right) * \gamma\right]$$

Figure 1: Example of a SAN model

automaton $\mathcal{A}^{(2)}$ is in state $1^{(2)}$, the transition from $0^{(1)}$ to $1^{(1)}$ state does not occur (rate equal to 0).

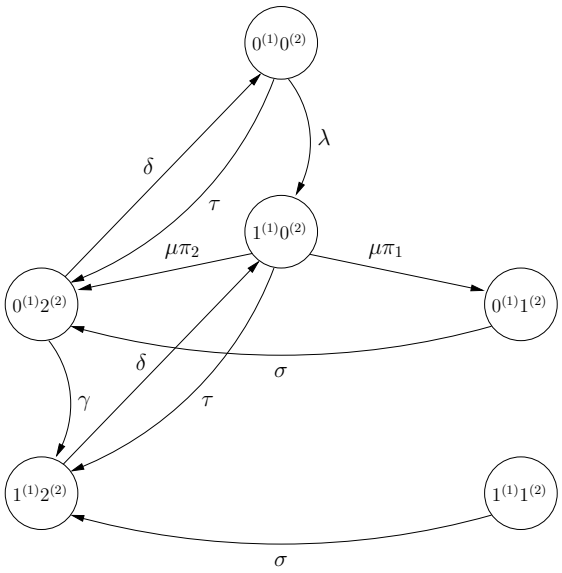


Figure 2: Equivalent Markov Chain to the SAN model in Figure 1

Figure 2 shows the equivalent MC model to the SAN model in Figure 1. It is important to notice that only 5 of the 6 states in this MC model are reachable. In order to express the reachable global states of a SAN model, it

is necessary to define a (*reachability*) function. For the model in Figure 1, the reachability function must exclude the global state $1^{(1)}1^{(2)}$, thus:

$$Reachability = ! \left[\left(st \ \mathcal{A}^{(1)} == 1^{(1)} \right) \&\& \left(st \ \mathcal{A}^{(2)} == 1^{(2)} \right) \right]$$

The use of functional expressions is not limited to event rates. In fact, routing probabilities also may be expressed as functions. The use of functions is a powerful primitive of SAN, since it allows to describe very complex behaviors in a very compact format. The computational costs to handle functional rates has decreased significantly with the developments of numerical solutions for the SAN models, *e.g.*, the algorithms for generalized tensor products [4].

3 Scheduling in NUMA OS

A system with shared resources needs to implement some policy to define who can use a specific resource. In an operating system, the process scheduler is responsible for managing the use of all system processors that are shared by processes. Scheduling in single-processor machines have been studied thoroughly in the past years. However, scheduling in multiprocessor machines still presents several challenges. Usually, shared memory multiprocessor machines can be classified as *Symmetric Multiprocessor* (SMP) or *Non-Uniform Memory Access* (NUMA). SMP machines are multiprocessor systems where each processor accesses any memory area in constant time. NUMA systems are multiprocessor systems organized in nodes. Each node has a set of processors and part of the main memory. The distance between nodes is not the same, hence there are different access times from each processor to different memory areas. Figure 3 shows an 8-processor NUMA machine organized in four nodes.

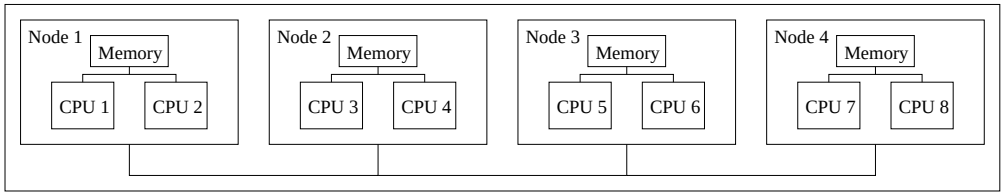


Figure 3: NUMA machine

3.1 Linux scheduler

One of the operating systems that implements a scheduler algorithm for parallel machines is Linux. Since version 2.5, the Linux scheduler has been called O(1) scheduler because all of its routines execute in constant time, no matter how many processors exist [15]. The current version of the Linux scheduler (kernel version 2.6.8) 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 its priority. The higher the process priority is, the higher will be 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 4 depicts part of this algorithm [15].

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

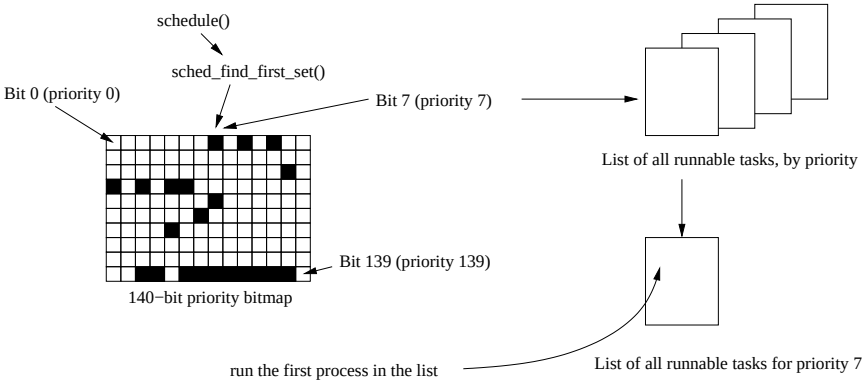


Figure 4: Selecting a new process to run

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 it 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.

When a process is created, it is inserted in the same runqueue and has 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 systems, 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 machines; 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 [5]. 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 system. Figure 5 shows *sched domains* created by Linux to the NUMA machine of Figure 3.

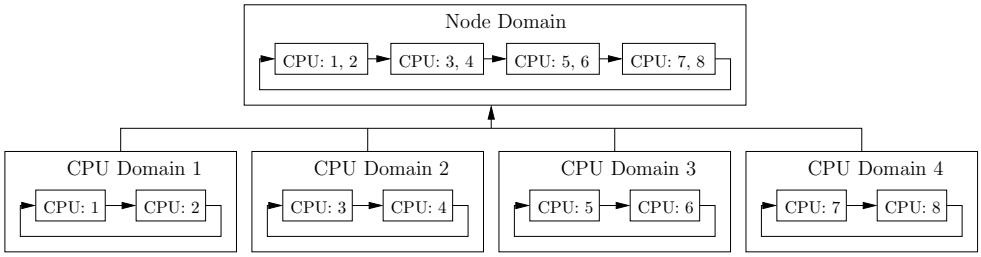


Figure 5: *sched domains* for a NUMA machine

The domains in the lowest level represent nodes of the system. These domains are called *CPU domains* because processes can be migrated only among CPUs, not among nodes. 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.

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 closer 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 node level only and an *exec* or *clone* system calls may not be invoked soon, periodical load balancing at regular intervals is also performed to prevent imbalances among the 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 current processor.

The choice of which 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.

4 Proposed Model

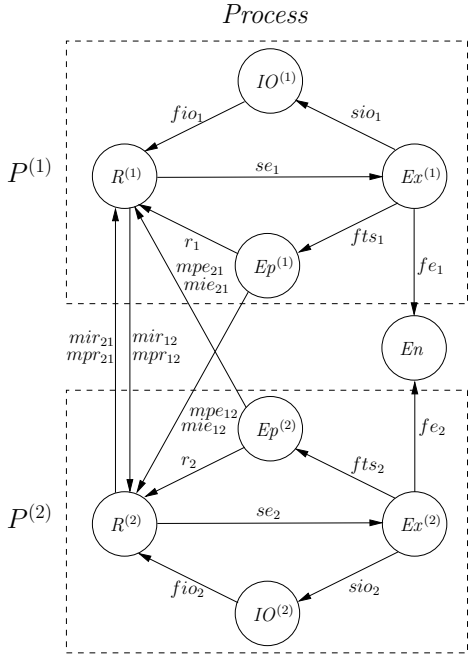
The use of benchmarks can be very useful to measure features in complex systems, *e.g.*, the Linux scheduling algorithm. However, as mentioned before, it can be very expensive to modify such systems in order to check them again and realize that all effort has been wasted. Instead, analytical modeling could be used to describe and evaluate those systems, and only if new modifications of the system turn out to be better than the previous one, actual implementation is realized. In this section we describe how the Linux scheduling algorithm can be modeled using the SAN formalism. The modular approach allowed by SAN is quite attractive to model such system due to its parallel behavior.

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. We propose a system model consisting of $P^{(i)}$ processors and one process.

4.1 Process

Figure 6 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; En representing that the process has finished its execution and it is not part of the system anymore.



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

Figure 6: Automaton *Process* and its events

It is important to notice that Figure 6 shows the behavior of the process in only two processors ($P^{(1)}$ and $P^{(2)}$). This was done to reduce the number of states in the figure for simplicity; 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.

The transitions represent the events that might happen during a process

lifetime. For instance, transition from $Ex^{(i)}$ to $Ep^{(i)}$ means that the process has finished executing its timeslice and will be stored in the expired queue. Some of the transitions represent the load balancing algorithm being executed. For example, transitions from $Ep^{(1)}$ or $R^{(1)}$ to $R^{(2)}$ represent that the process was in one of the queues from processor 1, and the load balancer chooses to move that process to the ready queue of processor 2. This moving represents that processor 1 was not balanced with respect to processor 2. The way the load balancer works was described in Section 3.

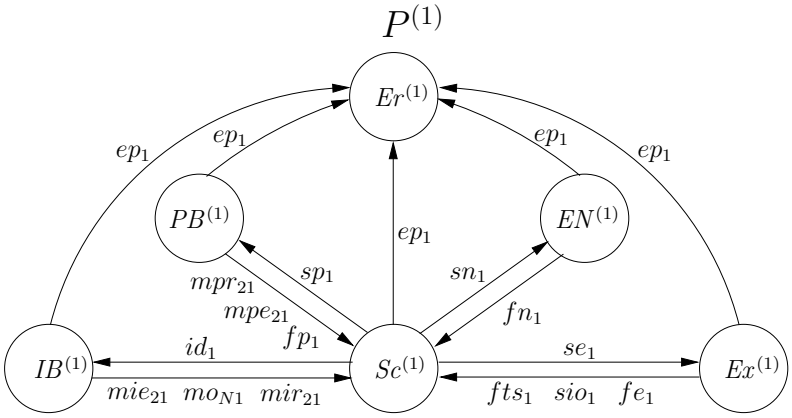
An important remark is that our approach allows different types of process configuration; if we want to analyse the behavior of an I/O-bound process, for instance, it is only necessary to adjust the rates appropriately. In this case, the transition rate from $Ex^{(i)}$ to $IO^{(i)}$ will be higher than the transition rate from $Ex^{(i)}$ to $Ep^{(i)}$ because the process will perform more I/O operations. As an I/O-bound process receives a greater priority than a CPU-bound process, the transition rate from $R^{(i)}$ to $Ex^{(i)}$ will increase as well. Besides process types (I/O-bound and CPU-bound), it is also possible to define the priority of a process.

As we mentioned before, each set $R^{(i)}$, $Ex^{(i)}$, $Ep^{(i)}$ and $IO^{(i)}$ is included for each processor in the system. It is necessary to have one $IO^{(i)}$ state for each processor modeled even though they represent exactly the same situation. If the *Process* automaton had only one global *IO* state, it would be impossible to know in which processor queue the process should be inserted.

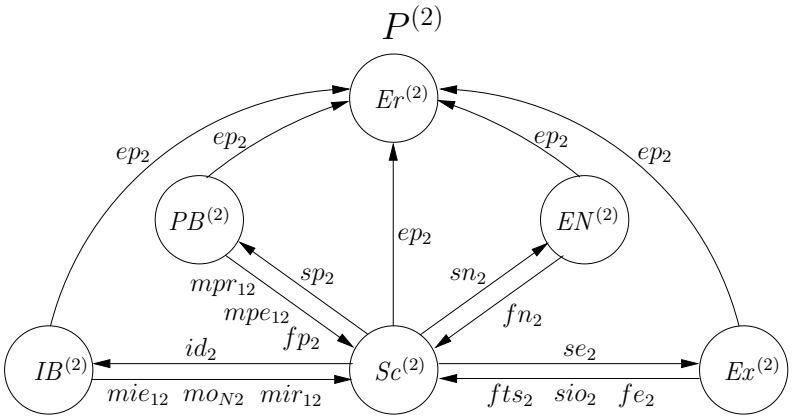
4.2 Processor

Figure 7 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 6; $Er^{(i)}$ representing that some error has occurred and the processor is not working.

It is important to remember that as we are describing a system in a NUMA machine, memory latency is different for each node. Memory latency is represented in our model by event rate fts and it can be different for each modeled processor. This allows to verify the system with different memory latencies values. For example, migrating processes from one node to another will cause



Type	Event	Rate	Type	Event	Rate	Type	Event	Rate
syn	mie_{21}	$r_{mie_{21}}$	loc	mo_{N1}	$r_{mo_{N1}}$	syn	mir_{21}	$r_{mir_{21}}$
syn	mpr_{21}	$r_{mpr_{21}}$	syn	mpe_{21}	$r_{mpe_{21}}$	loc	fp_1	r_{fp_1}
loc	id_1	f_{id_1}	loc	sp_1	r_{sp_1}	loc	ep_1	r_{ep_1}
loc	sn_1	r_{sn_1}	loc	fn_1	r_{fn_1}	syn	se_1	r_{se_1}
syn	fts_1	r_{fts_1}	syn	sio_1	r_{sio_1}	syn	fe_1	r_{fe_1}
syn	mie_{12}	$r_{mie_{12}}$	loc	mo_{N2}	$r_{mo_{N2}}$	syn	mir_{12}	$r_{mir_{12}}$
syn	mpr_{12}	$r_{mpr_{12}}$	syn	mpe_{12}	$r_{mpe_{12}}$	loc	fp_2	r_{fp_2}
loc	id_2	f_{id_2}	loc	sp_2	r_{sp_2}	loc	ep_2	r_{ep_2}
loc	sn_2	r_{sn_2}	loc	fn_2	r_{fn_2}	syn	se_2	r_{se_2}
syn	fts_2	r_{fts_2}	syn	sio_2	r_{sio_2}	syn	fe_2	r_{fe_2}



$$f_{id_1} = \left[(st\ Process\ != R^{(1)})\ \&\&\ (st\ Process\ != Ep^{(1)})\right] * r_{id_1}$$
$$f_{id_2} = \left[(st\ Process\ != R^{(2)})\ \&\&\ (st\ Process\ != Ep^{(2)})\right] * r_{id_2}$$

Figure 7: Processors automata and their events

the process to have different memory access times. This will be very important to check whether migrating processes can improve the system performance. If a processor is idle and another one is overloaded, it is probably better to move some processes even if it will take longer to access its data from memory. Our model can show what are the types of systems in which migration is better than leaving a processor overloaded, or vice-versa.

4.3 Events

In the previous sections, we have shown how a process and processors were modeled using SAN. This section lists all possible events that change the automata states:

- sio_i - the process is going to perform some I/O operations;
- fio_i - the process has finished its I/O operations and it has been moved to the ready queue in the i^{th} processor;
- se_i - the process has been scheduled in the i^{th} processor;
- fts_i - the process has finished its timeslice in the i^{th} processor;
- r_i - the process has been "moved" to the ready queue in the i^{th} processor;
- fe_i - the process has finished its execution;
- mpe_{ij} - the process was in the expired queue in the i^{th} processor and it has been migrated to the j^{th} processor by the periodical load balancing;
- mie_{ij} - the process was in the expired queue in the i^{th} processor and it has been migrated to the j^{th} processor by the idle load balancing;
- mpr_{ij} - the process was in the ready queue in the i^{th} processor and it has been migrated to the j^{th} processor by periodical load balancing;
- mir_{ij} - the process was in the ready queue in the i^{th} processor and it has been migrated to the j^{th} processor by the idle load balancing;
- ep_i - the i^{th} processor has failed;
- mo_{Ni} - other processes have been migrated through idle load balance;
- sp_i - periodical load balancing is going to be performed;

- fp_i - periodical load balancing was performed but did not affect *Process*;
- id_i - the scheduler could not find a process to schedule;
- sn_i - the scheduler algorithm has chosen another process to execute;
- fn_i - some other process has finished its timeslice or its execution.

4.4 Performability

With respect to the performability indices, we included the $Er^{(i)}$ state to represent a fault in a processor. Our fault model considers only fail-silent behavior, *i.e.*, when the processor fails, it will not produce any result and will stay in that state forever. Because we can have several processors in the system, we are interested in the situation in which the system continues working normally. Our aim is to verify the behavior of the system in the presence of faults (performability) [17]. One consequence of this approach is that we cannot calculate the stationary solution of the model because of the absorbent states ($Er^{(i)}$). The *Process* automaton has the same characteristic. The *End* state is an absorbent state. A possible solution to avoid absorbent states is to assume that the processor can be fixed, returning to its normal operation, and the process could start its execution again. In this paper we consider that a process executes once and that when a processor fails it cannot execute any other process.

4.5 Assigning Parameters

This section shows the numerical values that were assigned to the event rates. Some parameters were taken from benchmarks, whereas others are Linux variable or constants, *e.g.*, timeslices values. The benchmark used was the LMBench [16], performed on a 4-processor Itanium2 computer and on a 12-processor HP Superdome. The Linux kernel version used was 2.6.8 (with the ia64 patch applied). We also applied another patch that adds extra scheduling information to the */proc* directory [14].

Using both LMBench and the patch mentioned above, we assigned the following values to event rates: r_{fts_i} - timeslice (from 10 to 300 ms); r_{sp_i} - balance interval (200 ms - default value in the kernel 2.6.8); r_{sn_i} - time to schedule (1 ms); r_{fn_i} - timeslice (from 10 to 300 ms); $r_{mpe_{ij}}$, $r_{mie_{ij}}$, $r_{mpr_{ij}}$, $r_{mir_{ij}}$, $r_{mo_{Ni}}$, and r_{fp_i} - migration rates (approximately 1 ms). According to some benchmarks results there is one process migration every 0.8 ms.

Furthermore, some rates and parameters are chosen according to an actual process implementation, *e.g.*, how often or how long the process performs I/O operation (r_{sio_i} and r_{fio_i} respectively), how many processes exist in the system (N), number of processors (NP), how often the process is scheduled ($r_{se_i} = 1 - r_{sn_i}$), time spent in expired queue ($r_{r_i} = timeslice * (N/NP)$), error rate ($r_{ep_i}$), how often the scheduler does not find a process and the processor becomes idle (r_{id_i}), and how long the process will run until it reaches the *En* state (r_{fe_i}).

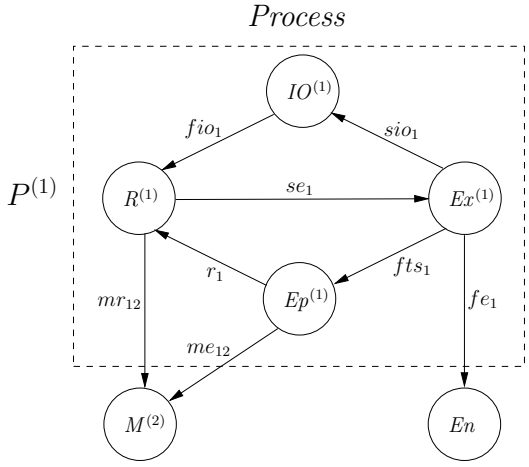
The r_{se_i} rate is defined as follow: according to the type of the process (I/O-bound or CPU-bound), its chance to execute (priority) is different. Hence, it is necessary to define how many processes have higher, lower or equal priority than the process modeled. So, the r_{se_i} rate is the sum of the probability of the lower priority processes and the exponentially distributed probability of the equal priority processes to execute.

5 Performance and Reliability Indices

As mentioned in Section 4, the proposed model is a generic approach that describes part of the Linux scheduling algorithm. Depending on the size of the system being modeled, the final automata could be quite big. However, it is possible to develop less complex models, which can be solved faster, based on the generic one. Hence specific performance and reliability indices can be obtained in a straightforward manner.

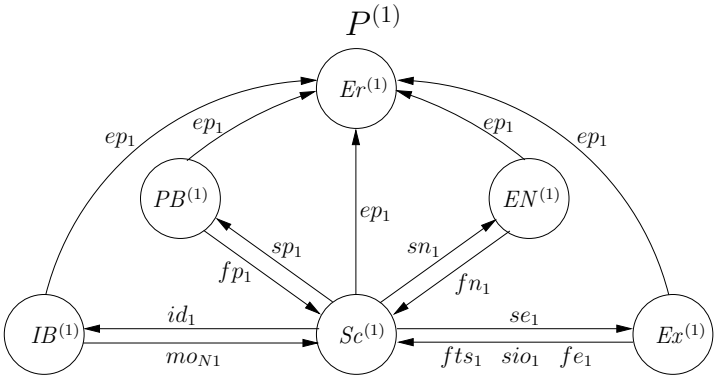
For instance, in order to obtain some migration information, such as *how long does it take to a process to migrate for the first time*, we can adapt the generic model. In the generic model (see Figure 6), the set of states $R^{(i)}$, $Ep^{(i)}$ $Ex^{(i)}$, and $IO^{(i)}$ represents the behavior of *Process* in the i^{th} processor. However, sometimes it is not necessary to model the behavior of *Process* in all processors. For this example, it is desirable to measure just migration information. In Figure 8, we present a reduced model that represents a *Process* in processor 1, and processor 2 is modeled as only one (absorbent) state ($M^{(2)}$).

In Figure 8, the $M^{(2)}$ state represents that *Process* have migrated from processor 1 to processor 2. Note that both synchronizing events mir_{12} and mpr_{12} become only one local event mr_{12}. Analogously, synchronizing events mie_{12} and mpe_{12} also become only one local event me_{12}. Such change occurs due to the fact that, as mentioned before, there is no need to model all processors. Therefore, those synchronizing events are not represented in the new processor



Type	Event	Rate	Type	Event	Rate	Type	Event	Rate
syn	sio_1	r_{sio_1}	loc	fio_1	r_{fio_1}	syn	se_1	r_{se_1}
syn	fts_1	r_{fts_1}	loc	r_1	r_{r_1}	syn	fe_1	r_{fe_1}
loc	me_{12}	$r_{me_{12}}$	loc	mr_{12}	$r_{mr_{12}}$			

Figure 8: Simplified automaton *Process*



Type	Event	Rate	Type	Event	Rate	Type	Event	Rate
loc	id_1	f_{id_1}	loc	sp_1	r_{sp_1}	loc	ep_1	r_{ep_1}
loc	sn_1	r_{sn_1}	loc	fn_1	r_{fn_1}	syn	se_1	r_{se_1}
syn	fts_1	r_{fts_1}	syn	sio_1	r_{sio_1}	syn	fe_1	r_{fe_1}
loc	mo_{N1}	$r_{mo_{N1}}$	loc	fp_1	r_{fp_1}			

$$f_{id_1} = \left[(st\ Process\ != R^{(1)})\ \&\&\ (st\ Process\ != Ep^{(1)})\right] * r_{id_1}$$

Figure 9: Simplified automaton $P(1)$ and its events

automaton (Figure 9).

However, if there are other processors in the system, there is no need to add new states to represent the new processors if their migration rates are the same. Otherwise, one state for each new processor must be created (different migration rates).

Note that this new approach reduces significantly the number of states in the model. In the generic model, the *Process* automaton has $(4 * NP) + 1$ states and the *Processors* automata ($P^{(i)}$) have $6 * NP$ states, whereas in the new model the *Process* automaton has $4 + (NP - 1) + 1$ states (for processors with different migration rates) or $4 + 1 + 1$ states (for processors with the same migration rate), and a single *Processor* automaton with 6 states. Any other model based on the generic one can cause bigger or smaller reduction.

5.1 Numerical Results

In this section, we have applied the analytical model presented in Section 4 to different NUMA machines. First of all, the results we obtained and present in the beginning of this section were applied to a 4-processor NUMA machine that is organized in four nodes and two memory access levels. Each node is composed of only one processor. A process executed in a processor (node) different from the one in which it was initially created will execute 25%[2] slower than it would in the processor in which it started its execution. Slower execution is due to time spent by the process to access its data, which is stored in a different node. Figure 10 shows a representation of this machine.

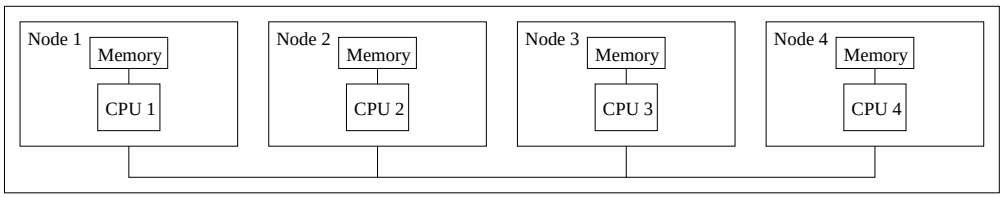


Figure 10: NUMA machine - two memory access levels

The Linux operating system creates, for the machine represented in Figure 10, the *sched domains* (see Section 3) showed in Figure 11.

Figure 12 shows the migration probability of an I/O-bound and a CPU-bound process. I/O-bound processes perform more I/O operations (IO state),

[2]We based this assumption on actual NUMA machine memory latency.

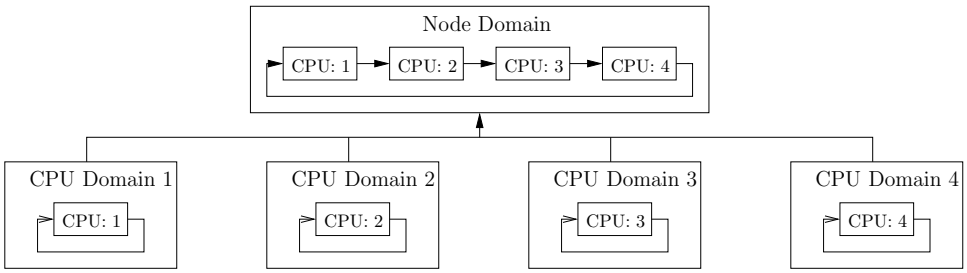
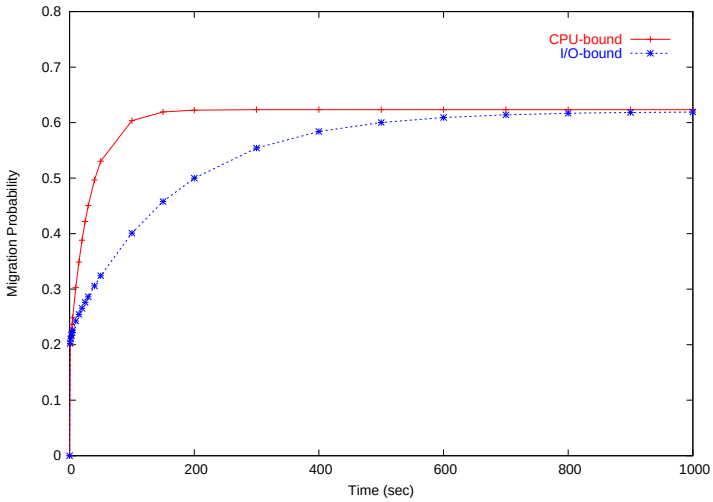


Figure 11: *sched domains* for the NUMA machine of Figure 10

while CPU-bound processes execute for a long time and go to the expired queue (*Ep* state) more frequently. As mentioned in Section 3, processes from the expired queue are preferred to be migrated to other processors. Therefore, a CPU-bound process tends to migrate faster than an I/O-bound process (see Figure 12).



Time	CPU	I/O	Time	CPU	I/O	Time	CPU	I/O
0	0.00000	0.00000	20	0.38817	0.26561	300	0.62332	0.55422
1	0.20098	0.20225	25	0.42188	0.27615	400	0.62337	0.58395
2	0.21220	0.21024	30	0.45075	0.28634	500	0.62337	0.60026
3	0.22430	0.21668	40	0.49667	0.30582	600	0.62338	0.60920
4	0.23649	0.22200	50	0.53038	0.32416	700	0.62338	0.61410
5	0.24844	0.22649	100	0.60355	0.40099	800	0.62339	0.61680
10	0.30271	0.24254	150	0.61914	0.45786	900	0.62340	0.61829
15	0.34880	0.25461	200	0.62246	0.49997	1000	0.62340	0.61911

Figure 12: Process migration behavior

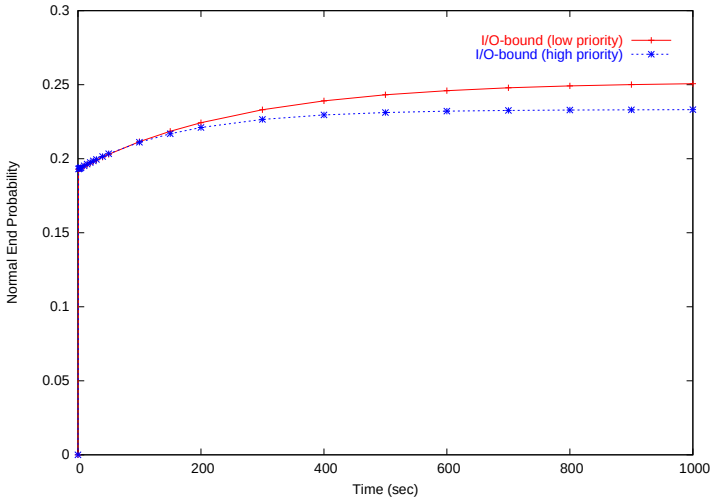
As an I/O-bound process spends more time performing I/O operations and less time in the expired queue, it takes longer to be moved to another processor. Such phenomenon occurs because I/O-bound processes tend to spend less time in the R and Ep states, consequently having less chance to be migrated. Note, however, that after a long period, I/O-bound and CPU-bound processes have a similar behavior. Such situation occurs because an I/O-bound process will receive the processor (go to Ex state) more frequently. Hence, in a long period, an I/O-bound process will finish its timeslice faster, therefore this type of process will be moved to the expired queue (Ep state) faster. This will result in a higher migration probability for an I/O-bound process after a long period of time.

Using the same model, we verified the normal end probability of an I/O-bound process without migrating to any other processor. Figure 13 shows the results for a low priority and for a high priority I/O-bound process. Although someone would expect a high priority process to finish first, observe that the low priority process has the highest normal end probability when migration is possible. It occurs because as the high priority process tends to execute more frequently, it also tends to migrate faster. This situation would not happen if processes could not be migrated.

Figure 14 presents the normal end probability of the generic model applied to the machine represented in Figure 10. In this model, we introduce the concept of fault in one or more processors. We have verified seven possibilities of faults:

- $K = 0$: all processors working;
- $K = 1$: processor 1 fails;
- $K = 1^*$: one processor fails, except processor 1;
- $K = 2$: processor 1 and 2 fail;
- $K = 2^*$: two processors fail, except processor 1;
- $K = 3$: processor 1, 2 and 3 fail;
- $K = 3^*$: processors 2, 3 and 4 fail.

We assume that a process was created in processor 1, *i.e.*, it executes slower in other processors (2, 3 or 4). As mentioned before, the process executes 25% slower in processors 2, 3 and 4 than in processor 1, due to memory latency.



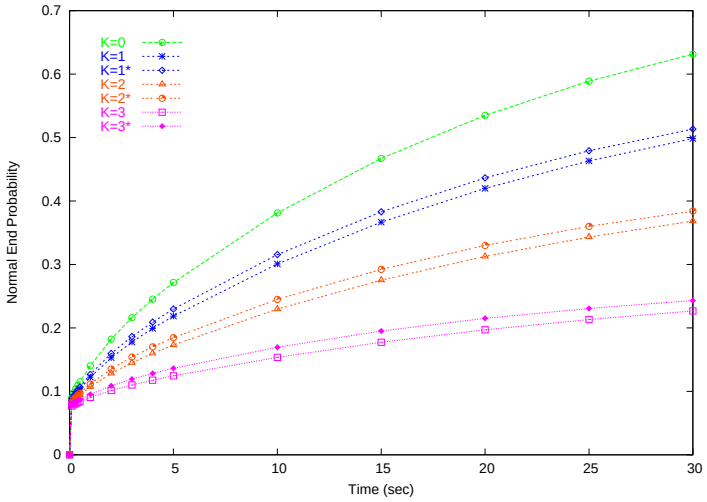
Time	Low	High	Time	Low	High	Time	Low	High
0	0.00000	0.00000	20	0.19707	0.19744	300	0.23309	0.22658
1	0.19284	0.19303	25	0.19810	0.19850	400	0.23908	0.22958
2	0.19311	0.19329	30	0.19912	0.19953	500	0.24318	0.23123
3	0.19335	0.19355	40	0.20109	0.20150	600	0.24598	0.23213
4	0.19359	0.19380	50	0.20300	0.20335	700	0.24789	0.23262
5	0.19382	0.19404	100	0.21149	0.21111	800	0.24920	0.23289
10	0.19493	0.19522	150	0.21851	0.21685	900	0.25009	0.23304
15	0.19601	0.19635	200	0.22432	0.22110	1000	0.25070	0.23312

Figure 13: End probability without migration

When a processor fails, its respective processes are moved to another processor. No process is lost.

The highest normal end probability ($K = 0$) indicates the best performance of the process, *i.e.*, all processors are working (see Figure 14). The system performance decreases as there are less processors working. It occurs due to the overload caused by the migration from the failed processor to the others. Observe that there are three distinct fault subsets ($K = 1$ and $K = 1^*$, $K = 2$ and $K = 2^*$, $K = 3$ and $K = 3^*$). In all cases, when processor 1 is working, the performance is better than when processor 1 has failed. Because the process was created in processor 1, it is normal that, when running in other processor, the process decreases its normal end probability.

As mentioned in the beginning of this section, we applied our model to different NUMA machines. The second machine configuration was used to compare the current Linux load balancing algorithm with a new strategy that is being developed in our research project. The machine is also a 4-processor



Time	$K=0$	$K=1$	$K=1^*$	$K=2$	$K=2^*$	$K=3$	$K=3^*$
0	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
0.1	0.08952	0.08479	0.08599	0.08097	0.08221	0.07688	0.07816
0.2	0.09781	0.09095	0.09265	0.08527	0.08704	0.07914	0.08097
0.3	0.10398	0.09556	0.09764	0.08852	0.09069	0.08085	0.08311
0.4	0.10961	0.09975	0.10218	0.09147	0.09401	0.08241	0.08505
0.5	0.11503	0.10378	0.10655	0.09429	0.09719	0.08388	0.08691
1	0.14028	0.12235	0.12668	0.10718	0.11170	0.09059	0.09531
2	0.18230	0.15301	0.15977	0.12825	0.13528	0.10144	0.10874
3	0.21617	0.17775	0.18631	0.14518	0.15405	0.11013	0.11931
4	0.24530	0.19915	0.20910	0.15983	0.17013	0.11765	0.12830
5	0.27161	0.21858	0.22968	0.17314	0.18463	0.12449	0.13636
10	0.38130	0.30077	0.31557	0.22970	0.24502	0.15367	0.16953
15	0.46713	0.36663	0.38298	0.27537	0.29237	0.17742	0.19506
20	0.53501	0.41996	0.43654	0.31269	0.32999	0.19699	0.21503
25	0.58879	0.46325	0.47924	0.34328	0.36003	0.21319	0.23074
30	0.63150	0.49848	0.51340	0.36845	0.38416	0.22666	0.24320

Figure 14: Process execution behavior

NUMA machine composed of four one-processor nodes. However, there are three memory access levels. In this type of machine a process, initially created in processor 1 (node 1), for instance, will execute 25% slower in processor 2 (node 2), and will execute 50% slower when executed in processor 3 (node 3) or processor 4 (node 4). Figure 15 represents this machine.

The current version of the Linux load balancing will create, for the machine in Figure 15, the same *sched domains* as those represented in Figure 11. Note that the three existing memory access levels are not represented in Figure 11. In order to better represent actual computer architectures, we propose to alter the Linux *sched domains* implementation to take several memory access levels

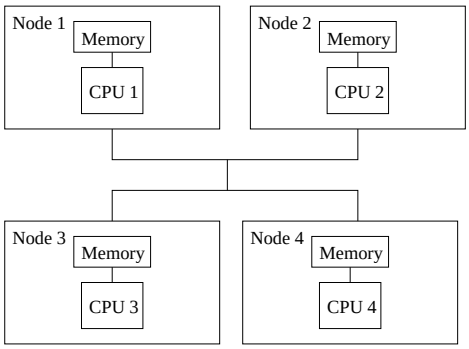


Figure 15: NUMA machine - three memory access levels

into consideration. In our proposal, the *sched domains* would be created as showed in Figure 16.

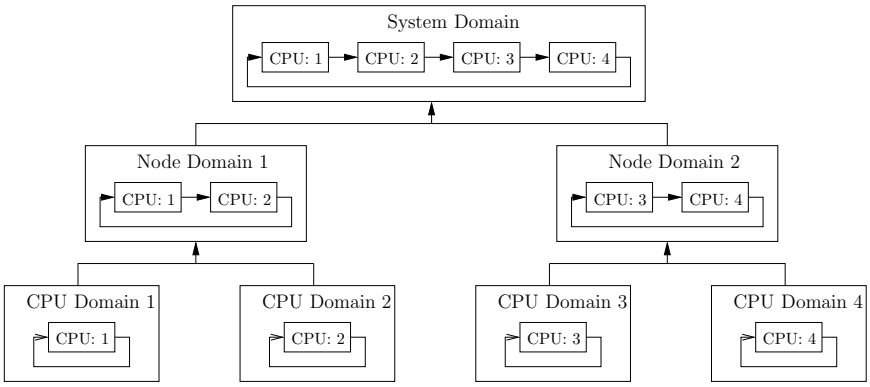
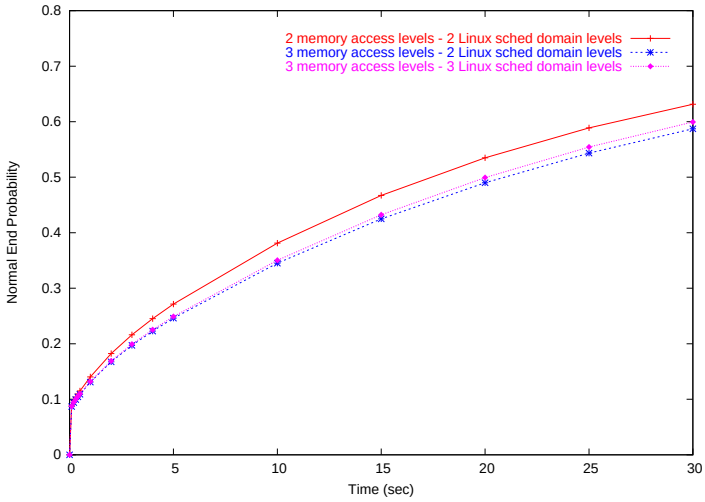


Figure 16: *sched domains* for the NUMA machine of Figure 15

Figure 17 shows the results obtained from both machines that we described in this section. The comparison between the two different machines is interesting because it shows that a machine with less memory access levels (*2 memory access levels - 2 Linux sched domain levels*) can have a better performance when the number of processors is limited. However, a machine with less memory access levels is not necessarily scalable, because the memory access bus can become a bottleneck.

Figure 17 also shows a comparison between the current load balancing performance and the proposal being developed in our research group. First we obtained the normal end probability of the machine in Figure 15 considering



Time	2 levels	3 levels 2 sched	3 levels 3 sched	Time	2 levels	3 levels 2 sched	3 levels 3 sched
0	0.00000	0.00000	0.00000	3	0.21617	0.19710	0.19921
0.1	0.08952	0.08704	0.08736	4	0.24530	0.22278	0.22533
0.2	0.09781	0.09422	0.09469	5	0.27161	0.24612	0.24912
0.3	0.10398	0.09957	0.10015	10	0.38130	0.34516	0.35041
0.4	0.10961	0.10443	0.10511	15	0.46713	0.42499	0.43238
0.5	0.11503	0.10912	0.10989	20	0.53501	0.49011	0.49941
1	0.14028	0.13097	0.13209	25	0.58879	0.54344	0.55436
2	0.18230	0.16747	0.16912	30	0.63150	0.58726	0.59952

Figure 17: Multilevel *sched domains* hierarchy

the current load balancing algorithm, *i.e.*, Linux does not represent all memory access levels in its *sched domains* hierarchy (*3 memory access levels - 2 Linux sched domain levels*). In this situation, there is an equal probability (33%) of migrating a process among processors. Finally, we altered our analytical model to represent the different distances among nodes (*3 memory access levels - 3 Linux sched domain levels*). The probability of a process to be moved to a closer processor will be higher (80%) than the probability of moving to a more distant processor (10% for each processor).

Although the improvement showed in Figure 17 for our proposal seems to be small, it is important to point out that for bigger machines this difference can become greater. We still have to apply our model to bigger machines[3].

[3]We have already studied a machine with 64 processors (32 nodes) in which our proposal will create 10 Linux *sched domain* levels.

6 Conclusion

This paper has presented an analytical model for the scheduling algorithm of the Linux operating system (kernel version 2.6.8). The main objective of this work is to show that analytical modeling can help in answering whether a possible modification in an algorithm should be implemented or not. We showed some of the results we obtained through the use an analytical tool, for example, probabilities of processes migration. This model was developed as part of a research project to modify the Linux operating system in NUMA computers. The main goal of this project is to make Linux more scalable. The model will help in providing Linux with a new load balancing strategy and new page migration for the Linux memory manager.

In order to model the parallel features existing in the operating system, we had to use a formalism that would allow us to express this parallelism. We studied several formalism to describe the Linux algorithms, and the one that seemed more attractive was the SAN formalism. Using SAN was very straightforward to describe parallelism in the Linux operating system. Maybe modelers with different backgrounds (*e.g.* SPN) could be more comfortable with other formalisms.

Even thought SAN has been used to describe the Linux algorithms, we used several benchmarks to obtain some of the rates needed for the analytical model. The main benchmark used was LMBench and some information provided in the */proc* directory of the Linux operating system. One of the next steps of this work is to implement the new load balancing strategy. We believe that this new strategy will be more scalable and better than the existing one (as showed in Section 5.1). Once this strategy is implemented, then new benchmarks results can be used to compare with the results obtained from the analytical model.

We did not tackle several issues related to the Linux scheduling, *e.g*, real-time issues. The Linux scheduler provides some facilities for realtime processes. The scheduling policies for realtime processes in the Linux operating system are different from ordinary processes. For example, a realtime process is never moved to the expired queue. Consequently, it is necessary to adjust our model (removing the $Ep^{(i)}$ state) to analyze realtime process behavior.

Generally speaking, we may summarize our contribution as an initial effort to describe a quite complex reality and extract performance and reliability indices. However, the obtained indices already can furnish useful information about the expected behavior of the Linux operating system. Such information is actually being used during the implementation of new load balancing strategies.

References

[1] AIM Technology. AIM Multiuser Benchmark - Suite VII. http://sourceforge.net/projects/aimbench/, 1996.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[3] A. Benoit, L. Brenner, P. Fernandes, and B. Plateau. Aggregation of Stochastic Automata Networks with replicas. *Linear Algebra and its Applications*, 386:111–136, July 2004.

[4] A. Benoit, L. Brenner, P. Fernandes, B. Plateau, and W. J. Stewart. The PEPS Software Tool. In *Computer Performance Evaluation / TOOLS 2003*, volume 2794 of *LNCS*, pages 98–115, Urbana, IL, USA, 2003. Springer-Verlag Heidelberg.

[5] 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.

[6] L. Brenner, P. Fernandes, and A. Sales. The Need for and the Advantages of Generalized Tensor Algebra for Kronecker Structured Representations. In 20^{th} *Annual UK Performance Engineering Workshop*, pages 48–60, Bradford, UK, July 2004.

[7] G. Ciardo and K. S. Trivedi. A Decomposition Approach for Stochastic Petri Nets Models. In *Proceedings of the* 4^{th} *International Workshop Petri Nets and Performance Models*, pages 74–83, Melbourne, Australia, December 1991. IEEE Computer Society.

[8] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK Benchmark: Past, Present, and Future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[9] P. Fernandes, B. Plateau, and W. J. Stewart. Efficient descriptor - Vector multiplication in Stochastic Automata Networks. *Journal of the ACM*, 45(3):381–414, 1998.

[10] G. Franks and M. Woodside. Multiclass Multiservers with Deferred Operations in Layered Queueing Networks, with Software System Applications. In 12^{th} *IEEE/ACM Internacional Symposium on Modelling, Analysis and Simulation on Computer and Telecommunication Systems (MASCOTS'04)*, pages 239–248, Volendam, The Netherlands, October 2004. IEEE Press.

[11] E. Gelenbe. G-Networks: Multiple Classes of Positive Customers, Signals, and Product Form Results. In *Performance*, volume 2459 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag Heidelberg, 2002.

[12] S. Gilmore, J. Hillston, L. Kloul, and M. Ribaudo. PEPA nets: a structured performance modelling formalism. *Performance Evaluation*, 54(2):79–104, 2003.

[13] L. Golubchik, J. C. S. Lui, E. de Souza e Silva, and H. R. Gail. Performance Tradeoffs in Scheduling Techniques for Mixed Workloads. *Multimedia Tools and Applications*, 21(2):147–172, 2003.

[14] Rick Lindsley. The Cursor Wiggles Faster: Measuring Scheduler Performance. In *Proceedings of the Linux Symposium*, volume 2, pages 301–310, Ottawa, Canada, July 2004.

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

[16] L. W. McVoy and C. Staelin. LMBench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference*, pages 279–294, San Diego, USA, January 1996.

[17] J. F. Meyer. Performability Evaluation: Where It Is and What Lies Ahead. In *The IEEE International Computer Performance and Dependability Symposium*, pages 334–343, Erlangen, Germany, April 1995. IEEE Computer Society Press.

[18] B. Plateau. On the stochastic structure of parallelism and synchronization models for distributed algorithms. In *Proceedings of the 1985 ACM SIGMETRICS conference on Measurements and Modeling of Computer Systems*, pages 147–154, Austin, Texas, USA, 1985. ACM Press.

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

[20] W. H. Sanders and J. F. Meyer. Stochastic Activity Networks: Formal Definitions and Concepts. In *Lectures on Formal Methods and Performance Analysis : First EEF/Euro Summer School on Trends in Computer Science*, volume 2090 of *Lecture Notes in Computer Science*, pages 315–343, Berg En Dal, The Netherlands, July 2001. Springer-Verlag Heidelberg.

[21] W. J. Stewart. *Introduction to the numerical solution of Markov chains*. Princeton University Press, 1994.

[22] J. Uniejewski. SPEC Benchmark Suite: Designed for Today's Advanced Systems. Technical Report 1, SPEC Newsletter 1, Fall 1989.

[23] W. Wei, B. Wang, and D. Towsley. Continuous-time hidden Markov models for network performance evaluation. *Performance Evaluation*, 49(1-4):129–146, 2002.