Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 151 (2006) 131–149
www.elsevier.com/locate/entcs

Analytical Modeling for Operating System Schedulers on NUMA Systems[1]

Rafael Chanin[2] Mônica Corrêa[3] Paulo Fernandes[4]
Afonso Sales[5] Roque Scheer[6] Avelino F. Zorzo[7]

Faculdade de Informática
Pontifícia Universidade Católica do Rio Grande do Sul
Porto Alegre, Brazil

Abstract

Performance evaluation by benchmarking is one of the main approaches for measuring performance of a computer system. However, it is important to measure parts of a system before they are even implemented. This can be achieved through an analytical description of the system, allowing the analysis of the system performance. Additionally, the analytical model can be extended to consider also reliability issues. This paper presents a generic model for an Operating System (OS) scheduler using the Stochastic Automata Networks (SAN) formalism. SAN are used to describe processes and processors in the OS and their behavior when processes have to be migrated. Moreover, processor failures are also modeled in order to provide reliability indices. The proposed model uses actual benchmarks results obtained from a 4-processor Itanium2 SMP machine and a 12-processor Itanium2 NUMA machine.

Keywords: performance evaluation, analytical models, stochastic automata networks, operating system scheduling.

1 Introduction

Even with the recent advances on grids and clusters, large shared memory computers are still required to solve some computational problems and run certain applications. These applications need scalable operating systems to provide them with an environment that can deal with all their computation needs. Usually the performance of such parallel systems is measured through benchmarking. The main

[1] This work was developed in collaboration with Hewlett-Packard Brazil R&D.

[2] Email: chanin@inf.pucrs.br
[3] Email: mcorrea@inf.pucrs.br
[4] Email: paulof@inf.pucrs.br (Corresponding author. Supported by CNPq/Brazil.)
[5] Email: asales@inf.pucrs.br
[6] Email: roque.scheer@hp.com (Roque Scheer is with Hewlett-Packard Brasil R&D.)
[7] Email: zorzo@inf.pucrs.br (Supported by CNPq/Brazil.)

doi:10.1016/j.entcs.2006.03.016

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 [18], SPEC Benchmark Suite [24], NAS Parallel Benchmarks [9] and LINPACK Benchmark [8]. The choice of benchmark depends on the features which 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 the 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 of the effort spent.

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 Markovian modelers [25,14]. Another valid option is the use of high level formalisms, such as Queueing Networks [11,12], which can provide insights about performance, but sometimes it assumes too unrealistic behaviors, *e.g.*, unlimited queues. Another possible solution is the use of structured formalisms [22,6,10,13] 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 [19], *e.g.*, indices related to the way the performance of a system is affected by 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) [21] formalism to describe performance and reliability indices of some of the Linux operating system algorithms. However, any other formalism, *e.g.*, *Stochastic Activity Networks* (SAN) [22], *Process Algebra* [13] and *Stochastic Petri Nets* (SPN) [6] 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 [10,2], 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 a multiprocessor machine. This generic model considers the possibility of faulty processors, process migration and process scheduling in the operating system. Furthermore, such a 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 present 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 obtained from our proposed model. Finally, Section 6 assesses future work and emphasizes the main contributions of this paper.

2 Stochastic Automata Networks

The SAN formalism was proposed by Plateau [20] 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 [21]. 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 the Markov Chain (MC) formalism [23,5]. The state of a SAN model, called *global state*, is defined by the cartesian product of the *local states* of all automata.

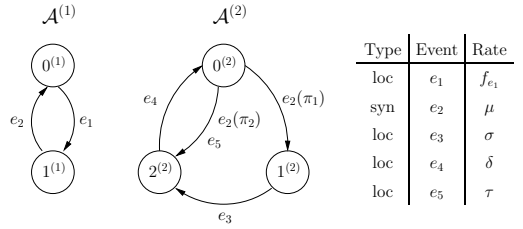
There are two types of events that change the global state of a model: *local events* and *synchronizing events*. Local events change the SAN global state passing from a global state to another that differs only by one local state. On the other hand, synchronizing events can change simultaneously more 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 the result of the occurrence of any number of events. In general, non-determinism (probabilistic choice) among possible different events is dealt with 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 provided. 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 syn-

chronizing events, functional rates are one-way interaction among automata, since it affects only the automaton in which it appears.

Fig. 1 presents a SAN model with two automata, four local events, one synchronizing event, and one functional rate. In the SAN model of Fig. 1, the rate of event e_1 is not a constant rate, but a functional rate f_{e_1} described by the SAN notation employed by the PEPS software tool [3]. The interpretation of a function can be viewed as the evaluation of an expression of non-typed programming languages, *e.g.*, the C language. Each comparison is evaluated to value 1 (*true*) or value 0 (*false*).



Type	Event	Rate
loc	e_1	f_{e_1}
syn	e_2	μ
loc	e_3	σ
loc	e_4	δ
loc	e_5	τ

$$f_{e_1} = [(st\ \mathcal{A}^{(2)} == 0^{(2)}) * \lambda] + [(st\ \mathcal{A}^{(2)} == 2^{(2)}) * \gamma]$$

Fig. 1. Example of a SAN model

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 [3].

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 has 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) [15]. SMP machines are multiprocessor systems in which each processor accesses any memory area in constant time. NUMA systems are multiprocessor systems organized in nodes. Fig. 2 shows an 8-processor NUMA machine organized in four nodes.

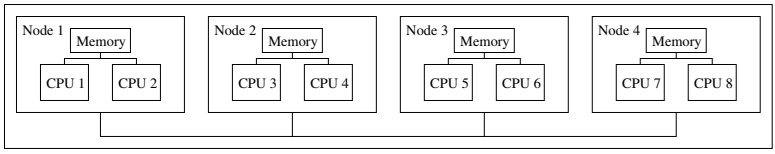


Fig. 2. NUMA machine

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.

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 [17]. 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 value 1. Then, the scheduler selects a new process to run by searching for the first bit equal to value 1 in the bitmap, which represents the highest priority of the runqueue, and finding the first process on the queue with that priority. Fig. 3 depicts part of this algorithm [17].

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 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.

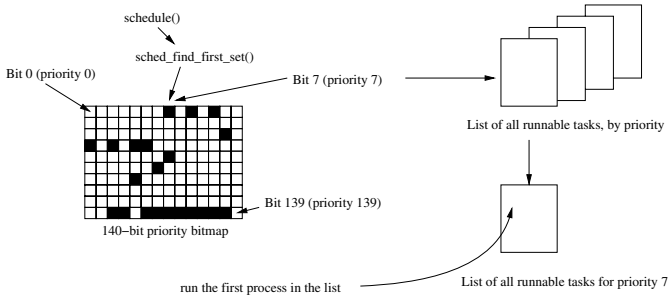


Fig. 3. Selecting a new process to run

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 situation, 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 processor with fewer 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 processor 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 [4]. 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. Fig. 4 shows *sched domains* created by Linux to the NUMA machine of Fig. 2.

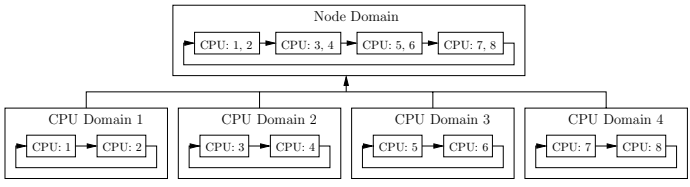


Fig. 4. *sched domains* for a NUMA machine

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 of 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 *exec* or *clone* system calls may not be invoked soon, periodical load balancing at regular intervals is performed to prevent imbalances among CPUs on different nodes. In this periodical load balancing, at each rebalance tick, the system checks if 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 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 of the effort has been wasted. Instead, analytical modeling could be used to describe and evaluate those systems, and only if new modifications of the system are predicted 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

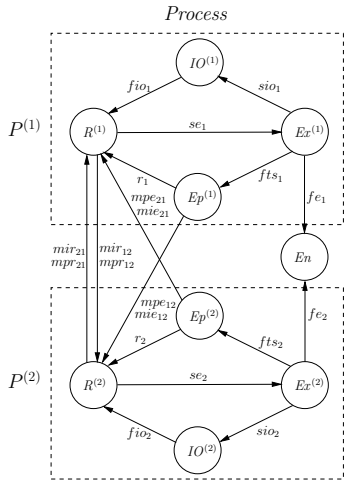
Fig. 5 shows the SAN model of a process in a 2-processor machine and its transition rate table. Table 1 shows all possible events that change the automata states.

Event	Description
sio_i	the process is going to perform an I/O operation
fio_i	the process has finished its I/O operation 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 the 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 the idle load balancing
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

Table 1
All possible automata events

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.

It is important to notice that Fig. 5 shows the behavior of the process in a computer with 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.



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

Fig. 5. Automaton *Process* and its events

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.

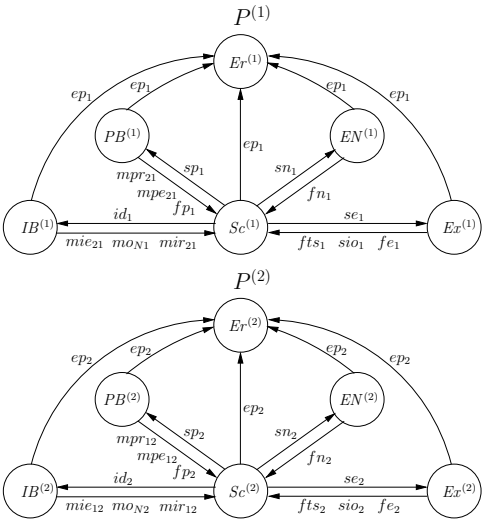
An important remark is that our approach allows different types of process configuration; if we want to analyze the behavior of an I/O-bound process, for instance, it is only necessary to adjust the rates appropriately. In this case, the transition rate from $Ex^{(i)}$ to $IO^{(i)}$ will be higher than 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 other process features, *e.g.*, process priority, process execution time, etc.

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, when the I/O operation finishes.

4.2 Processor

Fig. 6 shows the modeled processors using SAN and the corresponding transition rate table.

A processor might be in one of the following states: $IB^{(i)}$ representing that the



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

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

Fig. 6. Processors automata and their events

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 Fig. 5; $Er^{(i)}$ 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 fe and it can be different for each modeled processor. This allows to evaluate the system with different memory latencies values. For example, migrating processes from one node to another will cause 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 the types of systems in which migration is better than leaving a processor overloaded, or vice-versa.

4.3 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) [19]. One consequence of this approach is that we cannot calculate the stationary solution of the model because of the absorbing states ($Er^{(i)}$). The *Process* automaton has

the same characteristic. The *En* state is an absorbing state. A possible solution to avoid absorbing states is to assume that the processor can be fixed, returning to its normal operation, and the process could start its execution again.

4.4 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.*, timeslice values. The benchmark used was LMBench [18], performed on a 4-processor Itanium2 computer and on a 12-processor HP Superdome. The Linux kernel version used was 2.6.11 (with the ia64 patch applied). We also applied another patch that adds extra scheduling information to the */proc* directory [16]. Using LMBench and Linux patches, we assigned the following values to event rates (all rates are given in milliseconds):

- r_{fts_i} and r_{fn_i}: $\frac{1}{timeslice}$. Linux considers a timeslice varying from 10 to 300 ms. In our models, we defined I/O-bound timeslice as 200 ms and CPU-bound timeslice as 100 ms;
- r_{sp_i}: $\frac{1}{200}$. Linux default value in kernel 2.6.11 for balance interval is 200 ms;
- r_{sn_i}: $1 - r_{se_i}$. Considering scheduling time as 1 ms. Because either the modeled process is selected to execute or any other process is scheduled to run, this value depends on the r_{se_i} value;
- r_{fp_i} and $r_{mo_{Ni}}$: $\frac{1}{0.8}$. Considering 0.8 ms to perform the load balancing algorithm. When this algorithm is executed, it does not mean that migration will occur. Migration will happen only if the system is not balanced;
- $r_{mpr_{ij}}$, $r_{mir_{ij}}$, $r_{mpe_{ij}}$ and $r_{mie_{ij}}$: $\frac{1}{0.8} * \frac{1}{\frac{N}{NP}}$. Considering N the number of processes and NP the number of processors in the system;
- r_{r_i}: $\frac{1}{timeslice * \frac{N}{NP}}$. The modeled process must wait until all other processes finish their timeslices before it moves to the $R^{(i)}$ state.

Besides, some rates and parameters are chosen according to an actual process implementation:

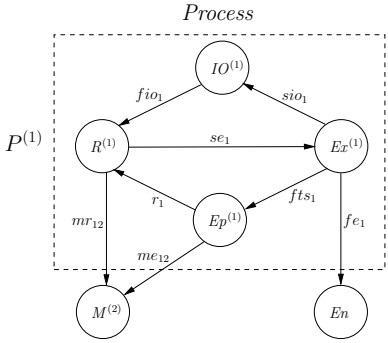
- r_{sio_i}: This rate must be greater than r_{fts_i} if the process is I/O-bound, otherwise it means that the process might be a CPU-bound one;
- r_{fio_i}: $\frac{1}{1{,}000}$. In almost all models, we considered that an I/O operation takes 1 second to be performed;
- r_{se_i}: $\frac{1}{0.2}$ for a CPU-bound process and $\frac{1}{0.7}$ for an I/O-bound process. It is necessary to define how many processes have higher, lower or equal priority than the modeled process. Therefore, the r_{se_i} rate is the sum of the execution probability of the lower priority processes and the exponentially distributed execution probability of the equal priority processes;
- r_{ep_i}: $\frac{1}{1}$. This rate is used for performability analysis, *i.e.*, when we want to analise the behavior of the system if one or more processors fail;

- r_{id_i}: $\frac{1}{10{,}000}$. In order to trigger the event for this rate, the modeled process cannot be either in the $R^{(i)}$ or in the $Ep^{(i)}$;
- r_{fe_i}: $\frac{1}{600}$. This rate is used to change the total lifetime of the process. Average duration of a process is calculated as $\frac{r_{fe_i}}{r_{sio_i}+r_{fts_i}} * timeslice$.

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. Therefore, 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 Fig. 5), 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 have just migration information. In Fig. 7, we present a reduced model that represents *Process* in processor 1, and processor 2 is modeled as only one (absorbing) state ($M^{(2)}$).



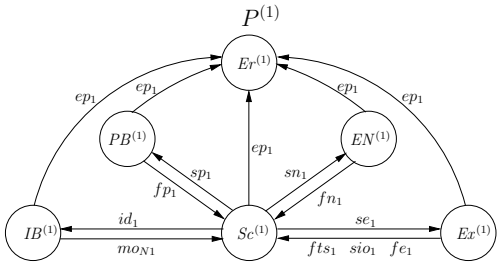
Type	Event	Rate
syn	sio_1	r_{sio_1}
syn	se_1	r_{se_1}
syn	fe_1	r_{fe_1}
syn	fts_1	r_{fts_1}
loc	fio_1	r_{fio_1}
loc	r_1	r_{r_1}
loc	me_{12}	$r_{me_{12}}$
loc	mr_{12}	$r_{mr_{12}}$

Fig. 7. Simplified automaton *Process*

In Fig. 7, state $M^{(2)}$ 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, 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. Hence, those synchronizing events are not represented in the new processor automaton (Fig. 8).

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



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

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

Fig. 8. Simplified automaton $P^{(1)}$ and its events

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.

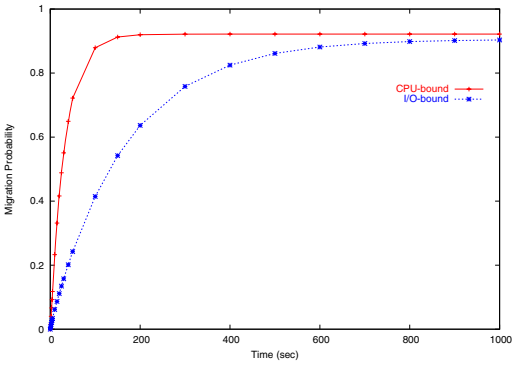
5.1 Numerical Results

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 at 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% slower [8] 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.

Fig. 9 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), 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 earlier than an I/O-bound process (see Fig. 9).

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 execute (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.

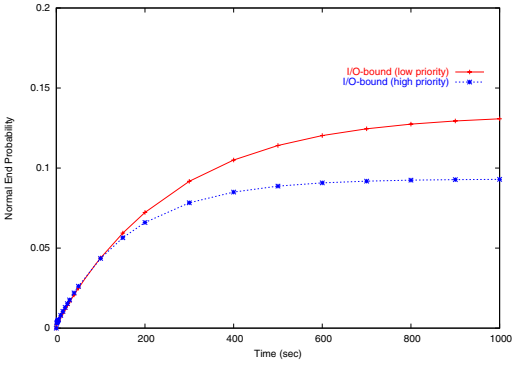
[8] We based this assumption on an actual NUMA machine memory latency.



Time	CPU	I/O	Time	CPU	I/O
0	0.00000	0.00000	50	0.72188	0.24230
1	0.01527	0.00722	100	0.87920	0.41462
2	0.04029	0.01463	150	0.91270	0.54217
3	0.06656	0.02146	200	0.91984	0.63660
4	0.09247	0.02785	300	0.92168	0.75824
5	0.11769	0.03390	400	0.92176	0.82489
10	0.23289	0.06140	500	0.92177	0.86142
15	0.33161	0.08675	600	0.92177	0.88143
20	0.41618	0.11109	700	0.92177	0.89240
25	0.48863	0.13465	800	0.92177	0.89841
30	0.55070	0.15750	900	0.92177	0.90171
40	0.64942	0.20117	1000	0.92177	0.90351

Fig. 9. Process migration behavior

Using the same machine, we verified the probability of an I/O-bound process to finish its execution (*normal end probability*) without migrating to any other processor. Fig. 10 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 earlier. This situation would not happen if processes could not be migrated.

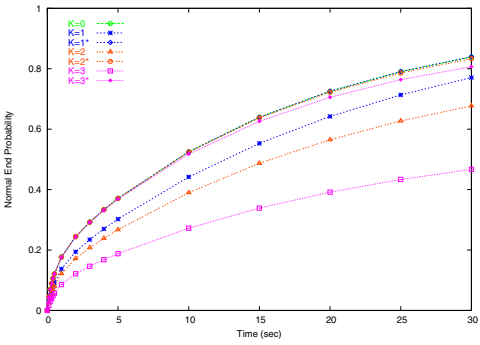


Time	Low	High	Time	Low	High
0	0.00000	0.00000	50	0.02499	0.02617
1	0.00231	0.00315	100	0.04383	0.04357
2	0.00301	0.00371	150	0.05940	0.05646
3	0.00366	0.00425	200	0.07228	0.06600
4	0.00417	0.00479	300	0.09172	0.07828
5	0.00467	0.00533	400	0.10501	0.08502
10	0.00711	0.00794	500	0.11409	0.08871
15	0.00950	0.01047	600	0.12030	0.09073
20	0.01184	0.01292	700	0.12454	0.09184
25	0.01414	0.01529	800	0.12744	0.09244
30	0.01639	0.01760	900	0.12942	0.09278
40	0.02077	0.02201	1000	0.13078	0.09296

Fig. 10. End probability without migration

Fig. 11 presents the normal end probability of the generic model applied to our example machine. In this model, we introduce the concept of fault in one or more processors. We have verified seven possible faulty scenarios: (i) $K = 0$ (all processors working); (ii) $K = 1$ (processor 1 fails); (iii) $K = 1^*$ (one processor fails, except processor 1); (iv) $K = 2$ (processor 1 and 2 fail); (v) $K = 2^*$ (two processors fail, except processor 1); (vi) $K = 3$ (processor 1, 2 and 3 fail); and (vii) $K = 3^*$ (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. When a processor fails, its respective processes are moved to another processor. No process



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	0.04437	0.02918	0.04437	0.02505	0.04436	0.01698	0.04436
0.2	0.06985	0.04896	0.06984	0.04253	0.06983	0.02926	0.06982
0.3	0.08932	0.06496	0.08931	0.05682	0.08929	0.03944	0.08927
0.4	0.10599	0.07884	0.10598	0.06924	0.10596	0.04832	0.10593
0.5	0.12079	0.09117	0.12077	0.08029	0.12074	0.05622	0.12069
1	0.17652	0.13750	0.17648	0.12169	0.17642	0.08574	0.17626
2	0.24444	0.19397	0.24440	0.17188	0.24425	0.12121	0.24374
3	0.29255	0.23450	0.29254	0.20768	0.29230	0.14627	0.29129
4	0.33368	0.26972	0.33372	0.23872	0.33337	0.16787	0.33178
5	0.37108	0.30226	0.37117	0.26734	0.37070	0.18772	0.36844
10	0.52486	0.44184	0.52523	0.38982	0.52392	0.27214	0.51748
15	0.63927	0.55310	0.63987	0.48718	0.63746	0.33868	0.62602
20	0.72515	0.64217	0.72593	0.56497	0.72224	0.39149	0.70552
25	0.78989	0.71349	0.79077	0.62718	0.78573	0.43351	0.76383
30	0.83888	0.77058	0.83980	0.67698	0.83342	0.46703	0.80664

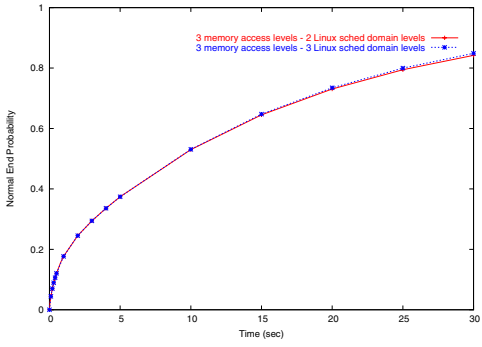
Fig. 11. Process execution behavior

is lost.

In the faulty scenarios $K = 0$, $K = 1^*$, $K = 2^*$, and $K = 3^*$, process performance is almost the same. As processor 1 does not fail, the process can execute faster in this processor because there is no need to migrate to another processor. However, in the scenarios $K = 1$, $K = 2$, and $K = 3$, performance decreases as there are less processors working. It occurs due to overload caused by migration from the failed processor to other processors. Besides, the process will run in a processor that is far from the process memory area. Hence, it will take longer to finish its execution.

As mentioned at the beginning of this section, we applied our model to different NUMA machines. In order to compare the current load balancing algorithm with a new strategy that is being developed by our research group, we applied our model to a 4-processor NUMA machine composed of four 1-processor nodes, but with three memory access levels.

The current version of the Linux load balancing algorithm creates, for this machine, two Linux *sched domain* levels. In order to better represent actual computer architectures, we have proposed to change the Linux *sched domains* implementation to take several memory access levels into consideration [7]. In our proposal, the Linux load balancing algorithm creates three Linux *sched domain* levels for this machine.



Time	3 levels 2 sched	3 levels 3 sched
0.1	0.04438	0.04438
0.3	0.08937	0.08937
0.5	0.12091	0.12091
1	0.17689	0.17687
3	0.29427	0.29417
5	0.37410	0.37407
10	0.52999	0.53093
15	0.64509	0.64755
20	0.73065	0.73473
25	0.79444	0.79997
30	0.84216	0.84885

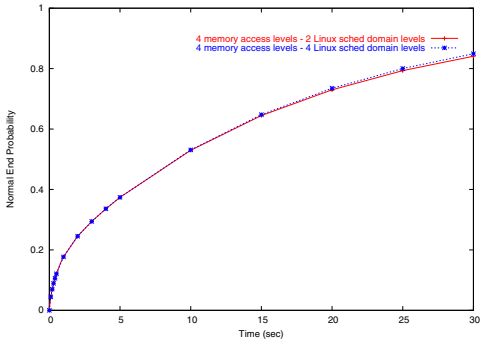
Fig. 12. 3-level NUMA machine

Fig. 12 shows the performance of a process running when Linux recognizes just 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 our approach presents a better performance depending on the time the process takes to execute (around 0.8% for a thirty second 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.

In order to verify whether our load balancing strategy would improve performance in machines with several memory access levels, we have applied our model to two other machine configurations:

a) four 1-processor nodes and four memory access levels: Fig. 13 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*). Compared to the first example (Fig. 12), 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 process to be migrated to the more distant node in the current Linux algorithm than in our approach.

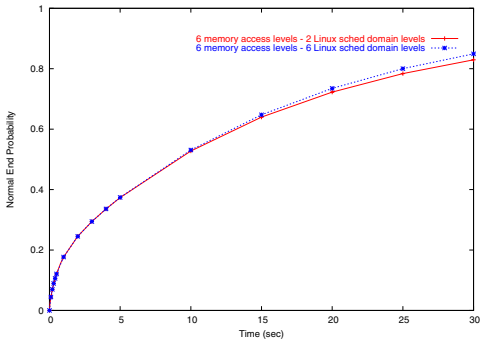


Time	4 levels 2 sched	4 levels 4 sched
0.1	0.04438	0.04438
0.3	0.08937	0.08937
0.5	0.12091	0.12091
1	0.17688	0.17687
3	0.29422	0.29416
5	0.37399	0.37407
10	0.52968	0.53098
15	0.64455	0.64768
20	0.72988	0.73495
25	0.79346	0.80027
30	0.84099	0.84921

Fig. 13. 4-level NUMA machine

b) six 1-processor nodes and six memory access levels: Fig. 14 shows the normal end probability of a process when Linux recognizes only two memory access levels (*6 memory access levels - 2 Linux sched domain levels*) and when Linux recognizes the actual computer's topology (*6 memory access levels - 6 Linux sched domain levels*). In this case the performance is better than in the other two models (around 2.4%). In this example, we have two more memory access levels than in the last example. Therefore, the chance for a process to be migrated to the more distant node in the current Linux algorithm is even higher.

Although the improvement showed in Fig. 12, 13 and 14 for our proposal seems to be small, it is important to point out that for bigger machines this difference can become greater. We can notice that as the number of levels increases, the difference between our approach and the current Linux algorithm increases. Besides, 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



Time	6 levels 2 sched	6 levels 6 sched
0.1	0.04437	0.04438
0.3	0.08937	0.08937
0.5	0.12091	0.12091
1	0.17688	0.17687
3	0.29416	0.29416
5	0.37361	0.37407
10	0.52753	0.53098
15	0.63991	0.64768
20	0.72269	0.73494
25	0.78398	0.80026
30	0.82962	0.84920

Fig. 14. 6-level NUMA machine

our results showed that the difference between our approach and the Linux one would increase as the time the process takes to finish increases. In addition, we still have to apply our model to bigger machines [9].

6 Conclusion

This paper has presented an analytical model for the scheduling algorithm of the Linux operating system (kernel version 2.6.11). 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 of 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 formalisms 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 confortable with other formalisms.

Although SAN has been used to describe the Linux algorithms, we used several benchmarks to obtain some of the rates for the analytical model. The main benchmark used was LMBench and some information provided in the */proc* directory of the Linux operating system. Based on the results obtained from our model, we have implemented a new version of the Linux load balancing algorithm [7] to take several memory access levels into consideration. We developed also a simulation model that showed similar results, for our load balancing strategy the simulation results showed an improvement of 2.5%. One next step is to compare the simulation and analytical results we obtained with actual results provided by benchmarks.

In this paper, we did not tackle several issues related to the Linux scheduling, *e.g*,

[9] We have already studied an actual machine with 64 processors (32 nodes) in which our proposal creates ten Linux *sched domain* levels.

realtime 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 processes.

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.

References

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

[2] 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.

[3] 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.

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

[5] L. Brenner, P. Fernandes, and A. Sales. The Need for and the Advantages of Generalized Tensor Algebra for Kronecker Structured Representations. *International Journal of Simulation: Systems, Science & Technology*, 6(3-4):52–60, February 2005.

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

[7] M. Corrêa, R. Chanin, A. Sales, R. Scheer, and A. F. Zorzo. Multilevel Load Balancing in NUMA Computers. Technical Report TR 049, PUCRS, Porto Alegre, 2005. http://www.inf.pucrs.br/tr/tr049.pdf.

[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] D. H. Bailey *et alli*. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[10] 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.

[11] G. Franks and M. Woodside. Multiclass Multiservers with Deferred Operations in Layered Queueing Networks, with Software System Applications. In *The* 12^{th} *MASCOTS*, pages 239–248, Volendam, The Netherlands, October 2004.

[12] E. Gelenbe. G-Networks: Multiple Classes of Positive Customers, Signals, and Product Form Results. In *Performance*, volume 2459 of *LNCS*, pages 1–16, 2002.

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

[14] 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.

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

[16] R. Lindsley. The Cursor Wiggles Faster: Measuring Scheduler Performance. In *The Linux Symposium*, pages 301–310, Ottawa, Canada, July 2004.

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

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

[19] 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.

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

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

[22] 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 *LNCS*, pages 315–343, Berg En Dal, The Netherlands, July 2001.

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

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

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