Performance Evaluation of Scheduling Policies in Symmetric Multiprocessing Environments

Jens Happe*, Henning Groenda*, and Ralf H. Reussner†
*FZI Forschungszentrum Informatik, Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, Germany
†Universität Karlsruhe (TH), Am Fasanengarten 5, 76131 Karlsruhe, Germany
Email: jhappe@fzi.de, groenda@fzi.de, reussner@ipd.uka.de

Abstract—**The shift of hardware architecture towards parallel execution led to a broad usage of multi-core processors in desktop systems and in server systems. The benefit of additional processor cores for software performance depends on the software's parallelism as well as the operating system scheduler's capabilities. Especially, the load on the available processors (or cores) strongly influences response times and throughput of software applications. Hence, a sophisticated understanding of the mutual influence of software behaviour and operating system schedulers is essential for accurate performance evaluations. Multi-core systems pose new challenges for performance analysis and developers of operating systems. For example, an optimal scheduling policy for multi-server systems, such as shortest remaining processing time (SRPT) for single-server systems, is not yet known in queueing theory. In this paper, we present a detailed experimental evaluation of general purpose operating system (GPOS) schedulers in symmetric multiprocessing (SMP) environments. In particular, we are interested in the influence of multiprocessor load balancing on software performance. Additionally, the evaluation includes effects of GPOS schedulers that can also occur in single-processor environments, such as I/O-boundedness of tasks and different prioritisation strategies. The results presented in this paper provide the basis for the future development of more accurate performance models of today's software systems.**

I. Introduction

Nowadays, multi-core processors with a symmetric multiprocessing (SMP) architecture are widely used in desktop systems and in server system. Such systems contain multiple identical physical processors with the same properties, e.g., memory access times and computation power. The broad introduction of multi-core processors poses new challenges for software performance evaluation. Software architects and developers need to determine the performance and scalability of their applications in SMP environments. In order to provide the necessary evaluation and/or prediction methods, it is necessary to understand the determining factors of multi-core processors for software performance first.

One of the major influences for software performance in SMP systems is the multiprocessor load-balancing policy implemented in general purpose operating system (GPOS) schedulers. Schedulers are responsible for distributing the system's load among the available processors. Load balancing is a multi-criteria optimisation problem in which certain trade-offs must be accepted. Common goals are high utilization of processors, and minimal response time for compute- as well as IO-bound or interactive tasks. Depending on the trade-off decision, load balancing policies implemented in GPOS strongly vary with respect to their effect on software performance.

In queueing theory, the influence of scheduling on performance has been intensively studied in [1]. It has been demonstrated that scheduling can affect response times by several orders of magnitude. Furthermore, the role of load distribution in multiprocessor systems became already evident [2], [3]. While the results provide the necessary theoretical background to gain a deeper understanding of load distribution, real operating system schedulers show a much higher complexity. For example, GPOS schedulers can dynamically redistribute the load of a system depending on a task's current or previous behaviour. Hence, a sophisticated understanding of the mutual influences of software behaviour and GPOS schedulers is essential for accurate performance evaluations.

In this paper, we aim for a deeper understanding of GPOS schedulers in SMP systems. We identify the critical factors and attributes of GPOS schedulers that determine software performance in a series of systematic experiments. For this purpose, we apply the Goal/Question/Metric (GQM) method [4]. In particular, we evaluate the influences of multiprocessor load balancing policies on software performance. Furthermore, we study the influences of time sharing, I/O-bound tasks, and different prioritisation strategies that are also relevant for single-processor systems. We specifically consider the operating systems Windows (Server 2003 and Vista) and Linux (Kernel 2.6.22).

The main contribution of this paper is an answer to the question: What are the relevant factors of GPOS schedulers that have to be considered in performance prediction? In particular, we identify the relevant influences of time sharing, scheduling of compute-bound, I/O-bound, and interactive tasks, different prioritisation strategies, and multiprocessor load balancing on software performance. The results of the experiments provide the necessary information to guide the future development of accurate prediction models for today's SMP systems.

This paper is structured as follows. In Section II, we describe the fundamental concepts of GPOS schedulers for SMP systems. The experimental method (GQM method) and the rationale of the experiments (GQM plan) are introduced in Section III. In Section IV, we describe the setup of the experiments. The results for single processor systems are described in Section V and for multiprocessor systems in

Section VI. In Section VII, we present the state-of-the-art of performance evaluation with respect to different scheduling policies. Finally, Section VIII concludes this paper.

II. Scheduling in Multiprocessing Environments

Modern operating systems can handle multiple processors and cores, including simultaneous multi-threading (SMT), symmetric multiprocessing (SMP), and non-uniform memory access (NUMA) architectures. Experience has shown that scheduling with per-processor run-queues largely outperforms per-system run-queues for usual desktop and server systems [5], [6]. However, relying on separate run queues for each processor requires a well-defined strategy to assign tasks to processors and to keep the load balanced. In the following, we describe the strategies implemented in the widely spread operating systems series Windows and Linux.

A. Windows

To optimise a task's performance, Windows always tries to assign one task to the same processor. This policy increases the probability of finding the task's data in the processor caches, which is likely to improve its computation speed. However, Windows aims to keep all processors busy at the same time. Each task receives an *ideal processor* during its creation following a simple round robin schema. This processor is always preferred during (re-)allocation. This strategy may require the task's insertion into the processor's run queue and, in some cases, the interruption of a running task. Only if the ideal processor is busy and other processors are idle, Windows looks for an appropriate new processor. Its first choice is the *last processor* the task has been running on (if not the same as the ideal processor). Next, it checks whether the currently active processor (i.e., the one performing the scheduling operation) is in the list of idle processors and tries to allocate the task there. If none of the above processors are idle Windows allocates the task to the first idle processor that is not sleeping and the task is allowed to be executed on. For SMT and NUMA architectures, the processor selection has to consider various other conditions, e.g., shared internal resources of a processor and memory access times, which we omit at this point.

The use of per-processor run queues requires a balance of the load among the available processors. If a processor is idle and its run queue is empty, the scheduler searches for an executable task and moves it to its run queue. This strategy avoids processors being idle while tasks are awaiting execution and keeps processor utilisation as high as possible.

B. Linux

The Linux scheduler tries to balance the system's load among all available processors and to assign fair shares of processing time to each task. For its load balancing decisions, Linux maintains a simplified model of the underlying processor architecture. Based on these hierarchical structures called *scheduling domains*, the scheduler decides when and where to move tasks.

For each scheduling domain, Linux determines how often to balance the processors, how much the load must differ to trigger balancing, and the time until cache affinity of a task is lost (i.e., the time a task is likely to find valid data in a processor cache). To maximise performance, Linux always selects tasks with the least cache affinity for moving. In this paper, we focus on Linux balancing capabilities in a single scheduling domain with SMP.

Linux balances SMP systems in regular intervals (called *active balancing*). It checks whether the processors in each domain require balancing. Active balancing ensures that processors executing multiple compute-bound tasks also participate in load balancing. The balancing interval determines how often balancing efforts occur. The interval grows if the system tends to stay in balance. For *event balancing*, the state change of a processor's run queue triggers load balancing. Possible events are the creation of a new task, the awakening of a task, or the removal of the last task in a processor's run queue (creating an idle processor).

III. Experiment Methodology

In the following, we describe the Goal/Question/Metric approach of Basili et al. [4] (Section III-A). Furthermore, we present the application of the GQM approach to design experiments for evaluating the influences of GPOS schedulers on software performance (Section III-B).

A. Goal/Question/Metric Approach

The Goal/Question/Metric (GQM) approach [4] is a process model for measurements targeting a particular set of issues (goals) and a set of rules for the interpretation of the measured data. In order to be meaningful, measurements must be goal-oriented and, thus, are defined in a top-down fashion. Basili et al. argue that measurements, which are not performed in a goal-oriented way, are likely to be inefficient. The absence of concrete goals carries the risk of collecting large amounts of unnecessary data. Large amounts of data and missing goals may complicate the interpretation of measurements. The GQM method provides a structured approach for the evaluation of operating system schedulers with respect to software performance.

The GQM method starts with the explicit definition of a measurement goal. Several questions serve to refine the goal and to identify its major components that need to be answered by measurements. Questions are further refined by metrics. When measurements for each metric have been taken, the resulting data is interpreted bottom up. Each metric is directed towards specific questions. The collected data answers the questions with respect to the goal. This evaluation allows deciding whether the goal has been attained or not.

B. GQM-Plan

Goal: The experiments that have been conducted in the scope of this paper serve as a basis for the design of realistic performance models. These models should capture the most relevant influences of GPOS schedulers and enable accurate performance predictions. Therefore, our experiments are directed towards the following goal:

> Identify the performance-relevant properties of operating system schedulers and quantify their effect on software performance from an application's point of view.

Questions: The goal is addressed by several questions that can be divided into two groups. The first group addresses performance-relevant properties of single processor scheduling, the second targets multiprocessor load balancing.

a) Scheduling of Single Processor Systems: In general, operating system schedulers implement a variant of round-robin scheduling (RR) to share a single processor among competing tasks. Each task is executed for a fixed time interval (called timeslice or quantum), pre-empted and the next task in the queue is executed. Depending on the operating system, the size of timeslices differs significantly. For example, Windows XP uses timeslices of 31 ms while Windows Server 2003 uses 187 ms. In software performance prediction, processor sharing (PS) is a common abstraction of RR. PS is an idealised approximation of RR where the duration of a timeslice as well as context switching times converge to zero. If n is the number of simultaneously running tasks, then each task receives $1/n$th of the processing power. However, experimental results, such as presented in [1], suggest that software performance can be very sensitive to modifications of the scheduling policy. For this reason, we ask: *How do RR and timeslices influence software performance?*

Furthermore, GPOS permit the assignment of priorities to tasks. If multiple tasks with different priorities compete for the processor, the operating system needs to implement a strategy to share the processor among these tasks. For example, Windows strictly prefers tasks with higher priorities. Thus, a task can only be executed if there is no task with a higher priority ready for execution. By contrast, Linux shares the processor among all task. The processing time, which a task receives, depends on its priority: the higher its priority the larger is its share of processing time. Thus, we ask: *How do priorities influence software performance?*

In order to keep the system responsive and the overall utilisation high, operating systems use heuristics that identify compute-bound, I/O-bound, and interactive tasks. The heuristics implemented in different operating systems can vary significantly. For example, Windows increases the priority of tasks that accessed resources (such as semaphores or network devices) for a specific time. By contrast, Linux tracks the past computation time and waiting time of a task. Depending on the ratio of both times it either increases or decreases the priority of a task. Due to such dependencies, we ask: *How does the OS influence the performance of I/O-bound and interactive tasks?*

b) Load-balancing in Symmetric Multiprocessing Systems: Today's operating systems implement very different strategies to balance the load among multiple processors (cf. Section II). While Linux tries to balance the load evenly, Windows tries to minimise the balancing effort while keeping all processors busy. Both strategies are likely to react differently under the same load conditions.

In the scope of this paper, we focus on two specific scenarios. In the first scenario, we minimise the number of events that trigger load balancing and, at the same time, impose a highly unbalanced load on the system. This scenario allows us to answer the question: *How does active, periodic load balancing influence software performance?*

In the second scenario, we introduce multiple events that can trigger load balancing and, again, start with a highly imbalanced load. This setting allows us to answer the following question: *How do events (e.g. wait operations) influence the load distribution?*

Metrics: To answer the questions presented above, we use the metrics response time, throughput, and load distribution. In the following, we give the definitions for these metrics used in the scope of this paper.

Response Time: A task's response time is highly sensitive to changes of the scheduling policies [1]. Therefore, it is a good indicator for possible performance influence of specific properties of operating system schedulers.

The response time of a task is the time span from the start of its computation to its completion. More formally, let T denote a finite set of tasks τ, $t_r \in \mathbb{R}_{\geq 0}$ one response time measurement of τ, and $\mathcal{P}(\mathbb{R}_{\geq 0})$ the power set of response time measurements, then the set of response time measurements of task τ is a function $\mathrm{RT} : T \rightarrow \mathcal{P}(\mathbb{R}_{\geq 0})$. For example, the set of response time measurements of some task τ_1 is $\mathrm{RT}(\tau_1) = \{1.5, 1.7, 1.3, 4.3\}$. Given a set of response time measurements, we can further define its arithmetic mean

$$\mathrm{E}[\mathrm{RT}(\tau)] = \frac{\sum_{t_r \in \mathrm{RT}(\tau)} t_r}{|\,\mathrm{RT}(\tau)|}$$

as well as all other statistical values, such as median, variance, or coefficient of variation (CV) [7]. Furthermore, $\mathrm{E}[\mathrm{RT}(T)] = \mathrm{E}[\bigcup_{\tau \in T} \mathrm{RT}(\tau)]$ denotes the average response time of all tasks $\tau \in T$.

Throughput: During our experiments, throughput is only of interest to determine the effect of priorities on performance. The share of time assigned to a task varies for different priorities. Thus, we can expect the throughput to change for tasks with different priorities. However, for all other scenarios, throughput plays a minor role since we have a low variance of job sizes and closed workloads [1].

Throughput is defined as the number of request completions during some fixed time interval. Let τ be the task of interest and M the time interval during which the response time measurements $\mathrm{RT}(\tau)$ have been taken. Then, we can simply define the throughput TP of τ as

$$\mathrm{TP}(\tau) = \frac{|\,\mathrm{RT}(\tau)|}{M}.$$

Load Distribution: The load distribution is good indicator for how "well" the load is distributed among the available processors. It allows to evaluate the effect of different load balancing strategies under different conditions.

In order to determine the load distribution, it is necessary to measure or estimate the load of individual processor cores. However, some operating systems (such as Windows Server 2003) do not support the measurement of a single processor's queue length. For other operating systems the granularity of measurements (e.g., the interval time when the queue length is updated) varies significantly. Since the underlying operating system must not influence the measurements, we propose the following heuristic approximation.

We estimate the load of a processor based on the response times of the currently running tasks. In our experiments, we chose computation times and workloads in such a way that PS represents a good approximation for the single processor scheduling. Having this in mind, we can compute the number of simultaneously running tasks by dividing the task response times by their raw computation time. For example, if a tasks processing time is 250 ms, the estimation computes a load of two tasks for a response time of 500 ms.

However, the execution of multiple tasks can overlap and a task's delay can further shift the overlap. Therefore, we cluster response times around multiples of the processing time. For the example above, we use a tolerance bound of $\pm$ 125 ms (half of the service demand) and, thus, consider all response times form 375 ms to 625 ms as concurrent execution of two tasks.

IV. Experimental Setting

In this section, we describe the setting of the experiments conducted in order to answer the questions posed in Section III-B. For this purpose, it is necessary to first clarify the assumptions underlying the experiments (Section IV-A). Then, we describe the hardware and software environment (Section IV-B) and introduce the load generator used in the scope of the experiments (Section IV-C).

A. Assumptions

The following assumptions define the scope of the experiments and ensure that the results are interpreted correctly.

Symmetric Multiprocessing Environments: In Section VI, we explicity focus on the performance influence of GPOS schedulers in SMP environments. In such systems, all processors and cores are similar with respect to their performance properties. We explicitly exclude processor architectures with specialized cores which can handle certain types of request faster than others (such as IBM's cell processor [8]). Furthermore, we exclude the influence of power-saving functionality that allows throttling the processing power in our experiments.

Excluding the Influence of Memory Access: In symmetric multiprocessing environments, different processors and cores may share common caches and memory buses. These low-level resources can become a limiting factor for the scalability of software applications. A task's memory usage, the processor's cache sizes, and the memory buses together determine the influence of memory accesses on software performance. Hence, the effect size varies for different processor architectures [9], [10], [11], [12]. The explicit exclusion of memory access keeps the evaluation largely independent of the chosen processor type (e.g., AMD X2 or Intel Core 2 Duo).

Limited Consideration of I/O Load: In our experiments, we focus on the effect of I/O-bound tasks on scheduling considering semaphores and network devices only. Both are typical I/O examples and affect the scheduling of tasks. However, they are not sufficient to reflect the influence of any I/O load on software performance in general. This focus avoids device-dependent effects such as caching effects of hard drives.

B. Hardware and Software Environment

The description of the experiment environment presented in this section should enable the reproduction of the experiment. It includes a description of the hardware and software environment, the implemented test driver and the measurement method.

Hardware: The experiments have been conducted on two different dual-core processors of different vendors: i) Intel Pentium D, 3 GHz, 2 GB RAM and ii) AMD Athlon 64 X2 Dual Core Processor 5200+, 2.61 GHz, 2 GB RAM. We also evaluated the influences of multiprocessor load balancing on a quad-core processor, an Intel Core 2 Quad Q6600 with 2.4 GHz and 2 GB RAM.

Software: On all three machines we installed the following operating systems: Windows Server 2003 (SP2), Windows Vista (SP1), and Ubuntu 7.10 Desktop Edition (Kernel 2.6.22). In all cases, we used Java to execute the experiments on different platforms. The used Java virtual machine was a Java HotSpot(TM) Client VM (build 1.6.0 03-b05, mixed mode, sharing).

C. Load Generator

Resource demands specified in the experiment scenarios need to be mapped to actual code that consumes the specified amount of processing time. We used algorithms like the Fast Fourier Transform or Fibonacci number computations to generate the necessary load. Such algorithms are, for example, used in the SPEC CPU2000 benchmark to measure the performance of a processor [13], [14]. Unfortunately, these benchmarks do not provide the information necessary to answer the questions specified in Section III-B and, thus, have to be adjusted for our purposes. Therefore, we implemented a resource demand generator [10] that automatically determines input parameters for an algorithm to meet the specified resource demands on a given platform. During a calibration phase, the dependency of input parameters and processing time is identified for an algorithm. The results are used to generate the request load during execution. The calibration measures the execution time of an algorithm in the single-threaded case, i.e., its (almost) uninterrupted and undisturbed execution time. During the experiment, the system may process multiple requests simultaneously. The measured performance metrics reflect influences of the underlying platform such as resource contention and caching effects. Thus, different load generating algorithms can lead to different performance results when executed simultaneously.

V. Performance Influences of Single Processor Scheduling

In this section, we answer the questions that address the influence of single processor scheduling on software performance (cf. Section III-B). The questions target the influence of RR scheduling and timeslices, task priorities, and I/O-bound and interactive tasks.

Timeslices and RR Scheduling: Most GPOS schedulers implement a variant of RR scheduling to share a single processor among multiple tasks. In software performance prediction, PS is an analytical approximation of RR. However, the quality of the approximation achieved by PS strongly depends on the examined scenario. In cases where the duration of a single task is smaller than a single timeslice, the prediction error can reach a multiple of the actual performance. To evaluate this dependency, we ran a simulation experiment for a simple example. A single server can process jobs either with RR, PS, or FCFS (first-come first-served). Jobs arrive in an open workload with a Poisson distribution and arrival rate $\lambda = 1/21ms$. The service time of each task is 20 ms (processing rate $\mu = 1/20ms$). These values result in an overall utilisation of $\lambda/\mu = 0.95$. The mean response times observed during the experiment are as follows:

Scheduling Policy	Mean Response Time
RR	231.5 ms
FCFS	254.7 ms
PS	488.3 ms

In this scenario, the response times for PS and RR differ by a factor of two. By contrast, the deviation of FCFS and RR is approximately 10%. The relation of requested computation time and timeslices results in a better approximation of RR by FCFS than by PS. While this effect is to be expected, the size of the deviation is not. Similar effects have been observed in [1]. However, if the service time of a task is much larger than a single timeslice, PS can provide a good approximation [15]. In real GPOS schedulers, timeslices are realised in various ways and their influence strongly depends on the considered scenario. Thus, timeslices should be modelled explicitly in software performance prediction.

Priorities – Fair and Unfair Time Sharing: In GPOS, different priorities can be assigned to tasks. In general, higher priority tasks are preferred over lower priority tasks. However, GPOS schedulers implement different strategies to share the processor among competing tasks with different priorities. Windows strictly prefers higher priority tasks over lower priority one. In other words, a task τ_l with priority p_l is only executed if there is no task τ_h with priority p_h and $p_h > p_l$. This strategy can lead to starvation of tasks with low priorities. By contrast, Linux assigns processing time to tasks according to their priority, i.e., the higher the priority of a task is the larger is its share of processing time.

A small experiment demonstrates the effect of varying timeslices on software performance under Linux. Its results are shown in Figure 1. In the experiment, two tasks with different priorities are executed concurrently in a closed workload

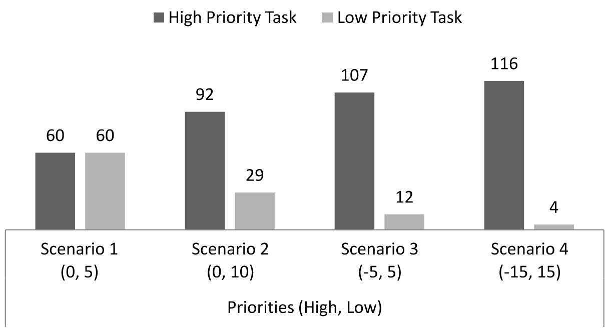


Fig. 1. Throughput [req/min] of task t_h and t_l with different priorities under Linux.

(without thinktime). Both tasks demand approximately 500 ms of processing time. If priorities are close to each other, both tasks receive the same share of processing time and, thus, experience a similar throughput (Scenario 1). The more the priorities differ, the larger becomes the difference of the throughput (Scenario 2 - 4).

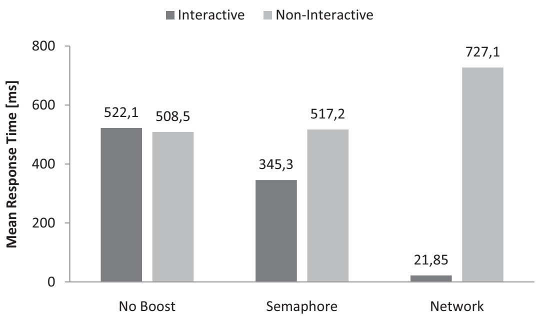
For Windows, the difference of priorities does not influence the performance of tasks τ_h and τ_l. As long as p_h is larger than p_l, τ_l is (nearly) starving and receives only very limited amount of processing time (60ms every 4000 ms) [6]. For these reasons, the influence of the implemented time sharing strategy should be modelled explicitly.

c) I/O-bound and Interactive Tasks: GPOS schedulers implement heuristics to optimise the overall system utilisation. In general, they favour I/O-bound and interactive tasks in order to keep the overall resource utilisation (besides the processor) high and to keep the system responsive at the same time. For this purpose, the scheduler needs to identify tasks that spend only a small fraction of time processing. Preferring these tasks over others can improve the overall system performance. Linux and Windows implement two inherently different strategies to reach this goal. In the following, we describe both strategies in more detail.

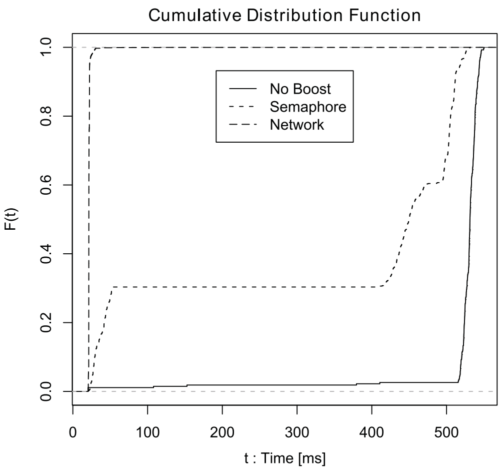
Windows implements a heuristic based on the resources used by a task. It increases (or boosts) the priority of a task whenever the task accesses a resource, e.g., acquires a semaphore or reads data from a network device. The priority bonus depends on the resource. If a task receives a bonus, its priority slowly decays back to its original value.

Figure 2 illustrates the effect of priority boosts on response time for the Windows operating system series. For the experiment, an open workload is generated, where an interactive (or I/O-bound) task τ_i and a compute-bound (or non-interactive) task τ_n arrive simultaneously. The resource demand of τ_i is 20 ms and its priority is $p_i = 8$. The resource demand of τ_n is 500 ms and its priority is $p_n = 9$. As long as τ_i is not boosted, it is delayed by τ_n. However, τ_i receives different boosts depending on the resources used.

Figure 2(a) shows how the resource used by τ_i affects its mean response time. If no resource is used (i.e., in the comparison scenario), τ_i is delayed by the complete execution time of τ_n which leads to a mean response time of approximately 520 ms. The cumulative distribution function (see Figure 2(b))



(a) Mean response times of tasks τ_i and τ_n.



(b) Response time of task τ_i.

Fig. 2. Mean value and cumulative distribution function for the task response times under Windows.

shows that this holds for all measurements. In the case of semaphore acquisition, task τ_i receives a priority bonus of 1 for (approximately) one timeslice. The bonus allows 30% of τ_i's requests to finish in about 20 ms while 70% are still delayed by τ_n. If a network connection is accessed, the priority bonus is 2 and lasts approximately 2 timeslices. Thus, τ_i can finish its request immediately leading to a mean response time of about 20 ms.

Linux implements a heuristic based on the time a task spends processing compared to the time it spends waiting. The relation of both processing times is mapped to priority bonuses or penalties between -5 and +5. A detailed evaluation of this strategy on software performance can be found in [16], [15]. In both cases, the authors observed that the scaling of dynamic priorities is not linear. A sharp jump of priorities from a penalty of 5 to a bonus of 5 occurs when a task spends at least 12% of its processing time waiting. This non-linear behaviour affects the performance of I/O-bound or interactive tasks as well as other tasks that are executed concurrently.

VI. Performance Evaluation of Multiprocessor Load Balancing

In this section, we evaluate the influence of multiprocessor load balancing on software performance in a controlled experiment. In the course of the experiments, we consider two different scenarios, which have been specifically designed to answer the questions posed in Section III-B. Furthermore, the experiments exclude disturbing effects of single processor scheduling (such as priority bonuses or penalties) as good as possible. In the following, we describe both scenarios in more detail.

A. Scenarios

With respect to multiprocessor load balancing, we are particularly interested in i) the influence of active (periodic) load balancing on software performance and ii) the influence of events (e.g. wait operations) on the load distribution. Therefore, we define two scenarios called `heavy load` and `moderate load` that address specific properties of both load balancing aspects. Scenario `heavy load` imposes continuous heavy load on all processor cores and, thus, leads to only few events that could trigger load balancing. In order to introduce events that can trigger load balancing, scenario `moderate load` introduces waiting times for each running task.

In both scenarios, multiple tasks are executed in a closed workload. Each task requests a specific computation time using the load generator described in Section IV-B. Table I summarises the computation time and think time for scenarios `heavy load` and `moderate load`. T_{Heavy} and $T_{Moderate}$ denote the set of tasks belonging to the respective scenario.

TABLE I
Description of the used closed workload scenarios.

task name	computation time t_c[ms]	think time t_w[ms]	tasks per processor
T_{Heavy}	250	0	(7/0)
$T_{Moderate}$	250	10	(7/0)

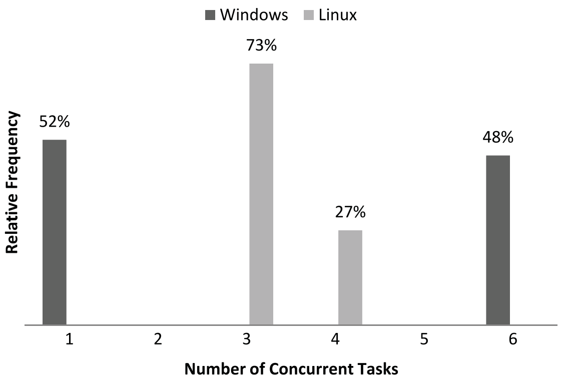
In order to allow the examination of load balancing effects on software performance, we enforced an intentionally imbalanced load distribution at the beginning of each experiment. We assigned all tasks to the first of the two available processors of the dual-core processor leaving the second processor idle (denoted by (7/0) in Table I).

B. Heavy Load Scenario

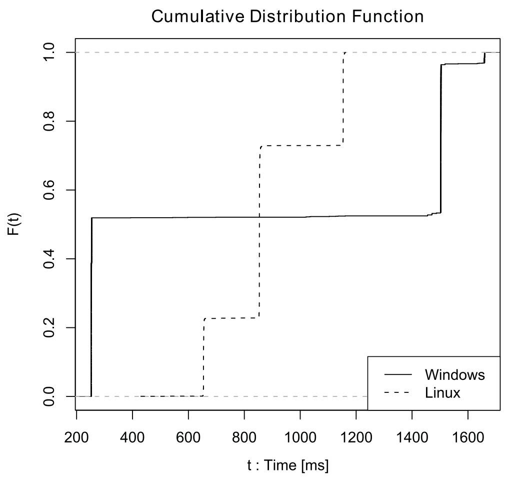
Figure 3 shows the measured response time and load distributions (cf. Section III-B) for the load balancing policies implemented in Linux 2.6.22 and Windows Server 2003. In the following, we describe the results of our experiments for both policies and discuss their differences.

Linux: In the observed load distribution for Linux (Figure 3(a), light gray), four tasks share one processor while the three remaining tasks share the other. Equation (Linux.1.a) generalises this observation for a system with n tasks and m processors:

$$\mathrm{Load}(\mathrm{CPU}_i) = \begin{cases} \lfloor n/m \rfloor & i > n \bmod m \\ \lceil n/m \rceil & i \leq n \bmod m \end{cases} \qquad \text{(Linux.1.a)}$$



(a) Scenario T_{Heavy}: Stable distribution of tasks to processors.



(b) Scenario T_{Heavy}: Response time distribution.

Fig. 3. Results of the experiments for scenario `heavy load`.

Linux targets an (almost) equal distribution of tasks among the processors. Equation (Linux.1.a) formalises the optimal balancing.

The corresponding task response time distribution (Figure 3(b)) has peaks at 650 ms, 850 ms, and 1150 ms. The timeslices of the Linux scheduler and the load distribution explain the peaks. In our experiment, we have two processor cores where one executes three and the other four tasks. Furthermore, the response time is split into two peaks for each processor. The distribution is a result of segmenting the computation time (denoted by t_c) on timeslices (denoted by TS). In order to explain the results, we need a function $\text{onCpu} : T \rightarrow \{\text{CPU}_1, \dots, \text{CPU}_m\}$ that returns the processor a task is assigned to. Equation (Linux.1.b) expresses the dependencies of timeslices and peaks for a task τ, $\text{Load}(\text{onCpu}(\tau))$, and timeslice TS:

$$\begin{aligned} min(\text{RT}(\tau)) = \; & \lfloor t_c / \text{TS} \rfloor * \text{Load}(\text{onCpu}(\tau)) * \text{TS} \\ & + (t_c \bmod \text{TS}) \\ max(\text{RT}(\tau)) = \; & min(\text{RT}(\tau)) \\ & + (\text{Load}(\text{onCpu}(\tau)) - 1) * \text{TS} \end{aligned} \qquad \text{(Linux.1.b)}$$

To determine the minimal response time ($min(\text{RT}(\tau))$) of task τ, the least number of preemptions during the computation ($\lfloor t_c / \text{TS} \rfloor$) is multiplied by the current load of the processor executing the task ($\text{Load}(\text{onCpu}(\tau))$) and the timeslice duration TS. The result includes the minimal time that τ spends waiting as well as the largest part of its processing time. Adding the remaining processing time ($t_c \bmod \text{TS}$) yields the minimal response time of τ. To compute the maximal response time ($max(\text{RT}(\tau))$), the waiting time for an additional preemption ($(\text{Load}(\text{onCpu}(\tau)) - 1) * \text{TS}$) is added to $min(\text{RT}(\tau))$.

For a computation time of $t_c = 250\,\text{ms}$, a timeslice of $\text{TS} = 100\,\text{ms}$, and a load of $\text{Load}(\text{onCpu}(\tau)) = 3$, the equation yields the response times $min(\text{RT}(\tau)) = 650$ and $max(\text{RT}(\tau)) = 850$. Furthermore, a load of $\text{Load}(\text{onCpu}(\tau)) = 4$ yields $min(\text{RT}(\tau)) = 850\,\text{ms}$ and $max(\text{RT}(\tau)) = 1150\,\text{ms}$. These results directly correspond to the observations in our experiment. In general, an average response time of $max(\text{RT}(\tau))$ occurs every $\text{TS} / (t_c \bmod \text{TS})$ requests and $min(\text{RT}(\tau))$ in the remaining cases.

d) Time Necessary for Load Balancing: Linux requires an initial balancing period to achieve a balanced situation. The balancing effort increases with a higher number of tasks. While Linux achieves a stable balance in 1.6 seconds for a system with three tasks and two processors, it requires more than 6 seconds for seven tasks.

Windows: For Windows (Server 2003 and Vista), the measured load distribution (Figure 3(a), dark gray) shows a larger imbalance compared to active-balancing. One task executes without interruption on the second processor while the remaining six tasks share the first processor. Equation (Windows.1.a) generalises this observation for a system with n tasks and m processors:

$$\text{Load}(\text{CPU}_i) = \begin{cases} n - m + 1 & i = 1 \\ 1 & \forall i \neq 1 \end{cases} \qquad \text{(Windows.1.a)}$$

The equation expects almost all tasks to be condensed on a single processor. The remaining processors execute a single task each. This behaviour is caused by Windows' load balancing policy that moves tasks only if a processor becomes idle. Thus, the initial imbalance (cf. Table I) remains during the whole experiment.

As to be expected from the load distribution, the response time distribution of all tasks (Figure 3(b)) shows two peaks at 250 ms and 1500 ms. In general, we can expect the following response times for a task τ with a computation time t_c in a system with a load of $\text{Load}(\text{onCpu}(\tau))$:

$$\text{E}[\text{RT}(\tau)] = \text{Load}(\text{onCpu}(\tau)) * t_c \qquad \text{(Windows.1.b)}$$

For $t_c = 250\,\text{ms}$ and $\text{Load}(\text{onCpu}(\tau)) = 1$ ($\text{Load}(\text{onCpu}(\tau)) = 6$), the equation yield the measured response times of $\text{E}[\text{RT}(\tau)] = 250\,\text{ms}$ ($\text{E}[\text{RT}(\tau)] = 1500\,\text{ms}$).

e) Decaying Load: Under Windows, load balancing is only triggered at the very beginning as the results in Section VI-B indicate. However, if the number of active tasks decreases over the duration of the experiment, load balancing is triggered whenever a processor becomes idle. In this case, Windows moves a task from the busiest processor to the idle one. For a system with n tasks and m processors at time t, the load distribution follows Equation (Windows.1.a).

Discussion of Scenario `heavy load`*:* The results of our experiments for the `heavy load` scenario demonstrate the differences between the the load balancing policies implemented in Windows and Linux with respect to their optimization goal. While Windows minimises the overhead and effort of load balancing, it accepts possibly large imbalances. By contrast, Linux balances the overall load distribution accepting possible costs for the additional load balancing effort. Windows targets an optimal utilisation of available processors. In our experiments, its load balancing policy resulted in a task distribution of (6/1). By contrast, Linux tries to equally balance the load on all processors. Its load balancing policy leads to a task distribution of (4/3) for the considered scenario.

C. Moderate Load Scenario

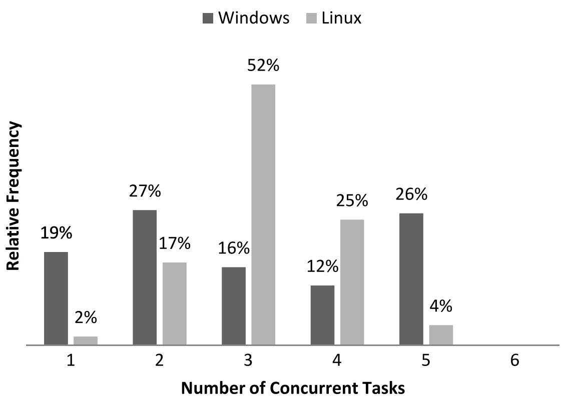
Figure 4 shows the load and response time distributions for scenario `moderate load`. Analogously to the previous section, we describe the results for Linux' and Windows' load balancing policies and discuss their differences.

Linux: For Linux, the measured load distribution (Figure 4(a), light gray) shows a larger variance compared to scenario `heavy load`. For 77% of all requests, the load is equally distributed between both processors as specified in Equation (Linux.1.a). However, the remaining 23% experience imbalances. This effect is caused by the waiting time introduced in scenario `moderate load`. The waiting time requires the continuous redistribution of tasks, since load changes continuously. Load balancing does not always react immediately on load changes. Furthermore, the number of active tasks varies during the experiment run. All these factors contribute to the observed imbalances. However, Equation (Linux.1.a) explains the observed behaviour for almost 80% of all requests.

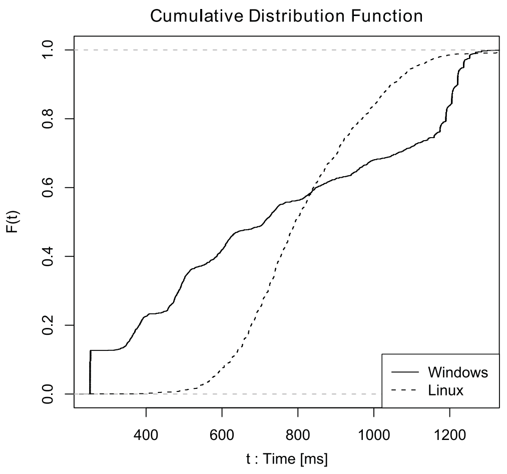
The combined response times of all tasks $\mathrm{RT}(T_{Moderate})$ (Figure 4(b)) lie in the interval ranging from 350 ms to 1150 ms. Compared to scenario `heavy load`, task response times show a larger and also softer distribution. Generally, the expected mean response times for a task $\tau \in T_{Moderate}$ depends on the number of tasks n and processors m as well as the computation time t_c (with $t_c > \mathrm{TS}$) and waiting time t_w (with $t_w < \mathrm{TS}$):

$$\mathrm{E}[\mathrm{RT}(\tau)] = \begin{cases} t_c & \text{if } n \leq m \\ t_c * n/m - t_w & \text{if } n > m \end{cases} \qquad \text{(Linux.2.a)}$$

In the first case, there are more processors than tasks. Thus, each task can proceed without interruption. In the second case, processing time is shared among tasks according to PS ($t_c * n/m$). Additionally, the task's waiting time has to be subtracted from its overall response time. This is necessary since all other tasks continue processing while τ is waiting. When τ returns, the currently running tasks already processed t_w of its timeslice. For scenario heavy load, Equation (Linux.2.a) yields



(a) Scenario $T_{Moderate}$: Stable distribution of tasks to processors.



(b) Scenario $T_{Moderate}$: Response time distribution.

Fig. 4. Results of the experiments for scenario `moderate load`.

$\mathrm{E}[\mathrm{RT}(\tau)] = 865\,\mathrm{ms}$. The measured waiting time (822.3 ms) deviates approximately 5% from the expectation.

The deviation of mean response times of individual tasks can give further insight into the capabilities of a load balancing policy. For active-balancing, each task received approximately the same share of processing time. The coefficient of variation (ratio of standard deviation σ and mean μ: $CV = \sigma/\mu$) of the mean response times of all tasks $\tau_1, \ldots, \tau_n \in T_{Moderate}$ is $CV(\{\mathrm{E}[\mathrm{RT}(\tau_1)], \ldots, \mathrm{E}[\mathrm{RT}(\tau_n)]\}) = 2.4\%$. The low CV points towards a balanced system.

Windows: The load distribution for Windows (Figure 4(a), dark gray) shows much stronger imbalances compared to Linux. The best approximation of an equal distribution (3/4) only occurs for 16% (three tasks) and 12% (four tasks) of all requests. In the most likely case, either two (27%) or five (26%) tasks are executed concurrently. Interestingly, 19% of all requests are processed without interruption. Their response time time is equal to the uncontended processing time of 250 ms. The expected counterpart of six concurrent tasks (distribution (1/6)) does not occur, since the waiting times of each task reduce the overall load.

The measured response times for all tasks of scenario

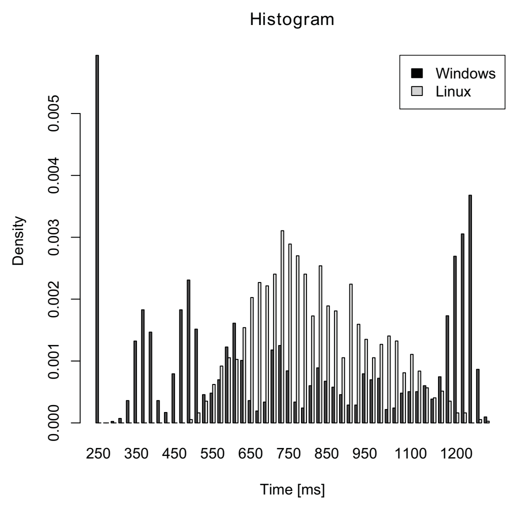


Fig. 5. Scenario $T_{Moderate}$: Response time distribution.

`moderate load` $\mathrm{RT}(T_{Moderate})$ are distributed between 250 ms and 1200 ms (cf. Figure 4(b)). The response times are grouped at several local peaks (see Figure 5). Using Equation (Linux.2.a) as an approximation of the mean response time of all tasks yields $\mathrm{E}[\mathrm{RT}(\tau)] = 865\,\mathrm{ms}$. The result deviates about 110 ms from the measurements. Analogously to Linux, the measured response time does not include initial queueing before the task is processed. Instead the time in the queue is assigned to the waiting time leading to a measured waiting time of 135 ms. For a waiting time of 135 ms, Equation Linux.2.a yields an expected mean response time of 740 ms. This result is much closer to the actually measured value of 755.5 ms (deviation $\approx$ 2%). For all n tasks $\tau_1, \ldots, \tau_n \in T_{Moderate}$, the coefficient of variation CV of set $\{\mathrm{E}[\mathrm{RT}(\tau_1)], \ldots, \mathrm{E}[\mathrm{RT}(\tau_n)]\}$ is 3.7%. The low coefficient indicates that each task receives the same amount of processing time on the long term.

Discussion of Scenario `moderate load`*:* In scenario `moderate load`, the load balancing policies of Windows and Linux can strongly affect the load distribution and response times of tasks. For both policies, the load is more distributed compared to the heavy load scenario (see Figure 4(a) and 3(a)). Even though Linux cannot achieve the optimal load distribution, it still keeps a better balance than Windows. For both policies, the task response times show a less regular distribution. However, the CV of the mean response times of all tasks is less than 5%. For scenario `heavy load`, the response times of single tasks varied by multiples. The reduced CV for scenario moderate load points towards a better balanced system.

VII. Related Work

In this section, we summarise existing work concerned with the evaluation of the performance influences of multiprocessor load balancing policies. We focus on work targeting general-purpose operating systems and briefly summarises results from the area of queueing theory.

General Purpose Operating Systems: Chanin, Correa et al. [17], [18] analysed the influence of different load balancing polices for Non-Uniform Memory Access (NUMA) systems on software performance focussing on the effect of different memory access times. They proposed an optimised multilevel load balancing algorithm and demonstrated with simulations, measurements and formal analyses of stochastic automata networks the possible performance gain of the new algorithm. However, the results of the simulation and formal analysis are contradicting. While the simulation and measurements yielded a performance gain of 2.2% to 10% depending on the underlying hardware architecture [18], the analytical results predicted an improvement of at most 1% [17]. The authors do not address the differences in the results of the predictions and measurements in the mentioned papers.

Kluge et al. [19] developed a framework for monitoring the Linux scheduler called VAMPIR that observes the number of task movements in multi-processor environments. In a larger case study, they observed the schedulers load balancing behaviour for a MPI application in three different scenarios. The results of Kluge et al. pointed out strong mutual dependencies between multiprocessor load balancing, task behaviour, and synchronisation methods. Different synchronisation methods and partitioning of the overall work into smaller blocks strongly affected the overall response times.

In [20], Bulpin and Pratt evaluate the performance of different SPEC CPU2000 benchmarks in an simultaneous multithreading (SMT) environment. They systematically executed different combinations of benchmarks concurrently. The results show that the actual performance gain or loss caused by the SMT technology strongly depends on the properties of the combined benchmarks. The observed effect ranges from a performance gain of more than 30% to a slowdown of more than 20%.

Single Processor and Multi Processor Scheduling in Queueing Theory: While scheduling policies for single-server systems (extended policies surveyed in [21]) are well studied and analytically tractable, multi-server queueing models pose several new challenges [22]. For example, the SRPT policy, which is proven to be the optimal scheduling policy with respect to mean response time for single-server queues, is not optimal for multi-server systems [23]. An optimal strategy for multi-server systems is yet unknown. Furthermore, analytical solutions have a limited availability, i.e., for specific combinations of scheduling and routing policies. Harchol-Balter et al. [24] analysed multi-server systems with prioritisation and compared the resulting response times with their single-server counterparts. They came to the conclusion that the effects of prioritisation in multi-server systems cannot be predicted by considering a comparable single-server system.

VIII. Conclusions

In this paper, we presented a series of systematic experiments to identify and quantify the performance-relevant factors of the schedulers implemented in Windows (Server 2003 and Vista) and Linux (Kernel 2.6.22). In our experiments, we identified different influences on software performance including: time sharing, prioritisation strategies, interactive and I/O-bound tasks, and multiprocessor load balancing.

For single-processor scheduling, the experiments demonstrated that the effect of simple round-robin scheduling on software performance depends on the type of workload and request sizes. Common performance models, such as processor sharing, can result in large prediction errors if utilisation is high and service times are smaller than a single timeslice. Furthermore, the heuristics to identify I/O-bound and interactive tasks affect the response time and throughput of tasks in a highly utilised system. The heuristics implemented in Windows and Linux behave entirely different, sometimes leading to unexpected behaviour [16], [15].

Multi-processor load balancing policies optimise performance with respect to different goals. Windows keeps the average utilisation high and tolerates major imbalances of the load. This policy can lead to a high variance in response times but achieves a good overall throughput. This effect becomes most evident in the `heavy load` scenario where the initial imbalances remain for the whole experiment. By contrast, Linux keeps an equally balanced system and accepts the higher costs for load balancing for the benefit of less variance in response time. Compared to Windows, this policy leads to a lower variance of response times for scenarios `heavy load` and `moderate load`.

The results presented in this paper provide a detailed evaluation of the performance influences of GPOS schedulers in symmetric multiprocessing environments. Currently, we are developing performance models for schedulers of SMP systems based on timed Coloured Petri Nets (CPN) [25]. The models include the influences of time sharing, priorities, I/O-boundedness, and multiprocessor load balancing. The CPN is hierarchically structured and comprises composable subnets for various parts of scheduling policies (e.g., multiprocessor load balancing). For design-time performance predictions, we plan to couple detailed scheduler models with behavioural models of the software system. Detailed models for common schedulers will be kept in a repository to reduce the complexity for software architects and performance analysts to a simple selection. Furthermore, we are evaluating and modelling the effect of scheduling in virtualised environments. Our scheduler performance model is designed in such a way that it can be extended towards virtualisation.

References

[1] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open versus closed: a cautionary tale," in *NSDI'06: Proceedings of the 3rd Symposium on Networked Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 239–252.

[2] W. Winston, "Optimality of the shortest line discipline," *Journal of Applied Probability*, vol. 14, no. 1, pp. 181–189, 1977.

[3] T. Osogami, "Analysis of multi-server systems via dimensionality reduction of markov chains," Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, USA, 2005.

[4] V. Basili, G. Caldiera, and H. Rombach, "The Goal Question Metric Approach," *Encyclopedia of Software Engineering*, vol. 1, pp. 528–532, 1994.

[5] J. Aas, "Understanding the Linux 2.6.8.1 Scheduler," Silicon Graphics, Inc. (SGI), Tech. Rep., 2005.

[6] D. A. Solomon and M. E. Russinovich, *Microsoft Windows Internals: Windows 2000, Windows XP und Windows Server 2003*. Microsoft Press, 2005.

[7] R. Jain, *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.

[8] IBM, "The Cell project at IBM Research," http://www.research.ibm.com/cell/, last retrieved 2008-08-16.

[9] J. Happe, H. Koziolek, and R. H. Reussner, "Parametric Performance Contracts for Software Components with Concurrent Behaviour," in *Proceedings of the 3rd International Workshop on Formal Aspects of Component Software (FACS)*, ser. Electronic Notes in Theoretical Computer Science, F. S. de Boer and V. Mencl, Eds., vol. 182, 2006, pp. 91–106.

[10] S. Becker, T. Dencker, and J. Happe, "Model-Driven Generation of Performance Prototypes," in *SIPEW 2008: SPEC International Performance Evaluation Workshop*, ser. Lecture Notes in Computer Science, vol. 5119. Springer-Verlag Berlin Heidelberg, 2008, pp. 79–98.

[11] E. Almog and H. Shachnai, "Scheduling memory accesses through a shared bus," *Performance Evaluation*, vol. 46, no. 2-3, pp. 193–218, 2001.

[12] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt, "QoS Policies and Architecture for Cache/Memory in CMP Platforms," *ACM SIGMETRICS Performance Evaluation Review*, vol. 35, no. 1, pp. 25–36, 2007.

[13] S. P. E. Corporation, "SPEC CPU2000 V1.3," http://www.spec.org/cpu/, 2000, last retrieved 2008-06-10.

[14] J. L. Henning, "SPEC CPU2000: measuring CPU performance in the new millennium," *IEEE Computer*, vol. 33, no. 7, pp. 28–35, 2000. [Online]. Available: http://dlib.computer.org/co/books/co2000/pdf/r7028.pdf

[15] J. Happe, "Predicting Software Performance in Symmetric Multi-core and Multiprocessor Environments," Dissertation, University of Oldenburg, Germany, August 2008.

[16] L. A. Torrey, J. Coleman, and B. P. Miller, "A comparison of interactivity in the Linux 2.6 scheduler and an MLFQ scheduler," *Software: Practice and Experience*, vol. 37, no. 4, pp. 347–364, 2006.

[17] R. Chanin, M. Correa, P. Fernandes, A. Sales, R. Scheer, and A. Zorzo, "Analytical Modeling for Operating System Schedulers on NUMA Systems," in *Proceedings of the Second International Workshop on the Practical Application of Stochastic Modeling (PASM 2005)*, ser. Electronic Notes in Theoretical Computer Science, vol. 151. Elsevier, 2006, pp. 131–149.

[18] M. Corrêa, A. Zorzo, and R. Scheer, "Operating System Multilevel Load Balancing," in *SAC '06: Proceedings of the 2006 ACM Symposium on Applied computing*. ACM, New York, NY, USA, 2006, pp. 1467–1471.

[19] M. Kluge and W. E. Nagel, "Analysis of Linux Scheduling with VAMPIR," in *Proceedings of the 2nd International Conference on Computational Science*, ser. Lecture Notes in Computer Science, vol. 4488. Springer-Verlag Berlin Heidelberg, 2007, pp. 823–830.

[20] J. Bulpin and I. Pratt, "Multiprogramming Performance of the Pentium 4 with Hyper-Threading," in *Proceedings of the Third Annual Workshop on Duplicating, Deconstruction and Debunking*, 2004, pp. 53–62.

[21] S. Aalto, U. Ayesta, S. Borst, V. Misra, and R. N. nez Queija, "Beyond Processor Sharing," *ACM SIGMETRICS Performance Evaluation Review*, vol. 34, no. 4, pp. 36–43, 2007.

[22] M. S. Squillante, "Stochastic analysis of multiserver systems," *SIGMETRICS Perfomance Evaluation Review*, vol. 34, no. 4, pp. 44–51, 2007.

[23] S. Leonardi and D. Raz, "Approximating Total Flow Time on Parallel Machines," in *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM, New York, NY, USA, 1997, pp. 110–119.

[24] M. Harchol-Balter, T. Osogami, A. Scheller-Wolf, and A. Wierman, "Multi-Server Queueing Systems with Multiple Priority Classes," *Queueing Systems: Theory and Applications*, vol. 51, no. 3-4, pp. 331–360, 2005.

[25] K. Jensen, L. M. Kristensen, and L. Wells, "Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 9, no. 3, pp. 213–254, 2007.