A Prediction Model for Software Performance in Symmetric Multiprocessing Environments

Jens Happe*, Henning Groenda†, Michael Hauck†, and Ralf H. Reussner‡
*SAP Research, Vincenz-Priessnitz-Strasse 1, 76131 Karlsruhe, Germany
†FZI Forschungszentrum Informatik, Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, Germany
‡Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany
Email: jens.happe@sap.com, groenda@fzi.de, hauck@fzi.de, reussner@kit.edu

***Abstract*—The broad introduction of multi-core processors made symmetric multiprocessing (SMP) environments mainstream. The additional cores can significantly increase software performance. However, their actual benefit depends on the operating system scheduler's capabilities, the system's workload, and the software's degree of concurrency. The load distribution on the available processors (or cores) strongly influences response times and throughput of software applications. Hence, understanding the operating system scheduler's influence on performance and scalability is essential for the accurate prediction of software performance (response time, throughput, and resource utilisation). Existing prediction approaches tend to approximate the influence of operating system schedulers by abstract policies such as processor sharing and its more sophisticated extensions. However, these abstractions often fail to accurately capture software performance in SMP environments. In this paper, we present a performance Model for general-purpose Operating System Schedulers (MOSS). It allows analyses of software performance taking the influences of schedulers in SMP environments into account. The model is defined in terms of timed Coloured Petri Nets and predicts the effect of different operating system schedulers (e.g., Windows 7, Vista, Server 2003, and Linux 2.6) on software performance. We validated the prediction accuracy of MOSS in a case study using a business information system. In our experiments, the deviation of predictions and measurements was below 10% in most cases and did not exceed 30%.**

I. Introduction

In the recent years, multi-core processors with a symmetric multiprocessing (SMP) architecture gained a large momentum. SMP systems consist of multiple identical physical processors (or cores) which share the same properties, e.g., memory access times and computation power. Business application development has changed due to this trend to increase the overall processing speed by parallel processing instead of faster processors. Architectural design decisions which affect size and number of tasks can have severe performance implications on different schedulers. Hence, there is a need of software architects and developers to determine the performance and scalability of their applications in these environments. For this purpose, reliable prediction models are essential.

Scheduling policies can affect the response time of an application by several orders of magnitude [21]. This is due to the fact that scheduling is a multi-criteria optimisation problem in which certain trade-offs must be accepted. In SMP environments, common goals are high utilisation of processors and minimal response times for compute, IO-bound, or interactive tasks. Existing prediction approaches (such as [6], [12]) rely on processor sharing (and its variants), preemptive priority scheduling and other abstract scheduling policies as described in [17], [25], [1] to approximate the behaviour of general purpose operating systems (GPOS). However, these abstractions are insufficient in SMP environments especially on applications using parallelisation on a fine-grained level. For multi-servers systems (including multi-core processors), various load distribution techniques are available [20], [19]. In contrast to GPOS schedulers, these policies do not take into account a task's current or previous behaviour when dynamically redistributing the load of a system. However, the policies' distribution of load on the available processors strongly influences response times and throughput of applications. The implications of not considering scheduling policies when designing software for SMP systems can range from negligible effects over wrongly dimensioned hardware resources to degraded performance compared to single-core system performance [8].

In this paper, we present a performance **M**odel for general-purpose **O**perating **S**ystem **S**chedulers (MOSS). MOSS is defined in terms of hierarchically structured, timed Coloured Petri Nets [10]. We use feature diagrams [5] to express customisations of MOSS that reflect the behaviour of different operating system schedulers. Furthermore, we developed a discrete event-simulation based on SSJ [14] to predict the effect of GPOS on applications. This simulation is integrated with a model-driven performance prediction framework (Palladio Component Model, PCM [3]). The combination of both approaches demonstrates the benefit of integrating MOSS into model-based prediction approaches. It enables software architects and performance analysts to evaluate their design with respect to different scheduler models and choose the best software design based on quantitative analysis. Integrated in the PCM, MOSS further supports the estimation of appropriate hardware needs of newly developed applications and helps to identify scalability bottlenecks and overloaded resources.

The contribution of this paper is MOSS, a validated model for performance predictions of software applications in SMP environments. MOSS reflects the influence of different schedulers including Windows 7, Vista, and Server 2003 as well as Linux (Kernel 2.6). To achieve this, we build our model on a series of systematic experiments that identify the determining

features of operating system schedulers [7]. Furthermore, we evaluated the prediction accuracy of MOSS in a case study using a business information system. In our experiments, the response time varied up to a factor of four depending on the chosen operating system. While such effects can hardly be captured by existing prediction approaches, MOSS was able to predict the system's response time with an error of less than 5% to 10% in most cases and did not exceed 30%.

This paper is structured as follows. First Section II provides an overview of MOSS. In Section III, we present performance-relevant load-balancing GPOS scheduler features. Section IV describes the couloured Petri net (CPN) models of MOSS and how they can be integrated into prediction approaches. In Section V, we evaluate the prediction accuracy of MOSS in a case study. Related work is presented in Section VI. Finally, Section VII concludes this paper.

II. Overview

The goal of MOSS is to provide a detailed model of operating system schedulers that allows accurate performance predictions in SMP environments. However, operating systems implement different strategies to balance the load among the available processors, to optimise for I/O bound and interactive tasks, and to share a single processor among multiple tasks. Thus, MOSS needs to reflect the effect of different strategies and their combinations to capture the behaviour of multiple operating systems. For this purpose, MOSS can be configured with different strategies for scheduling and multiprocessor loadbalanding.

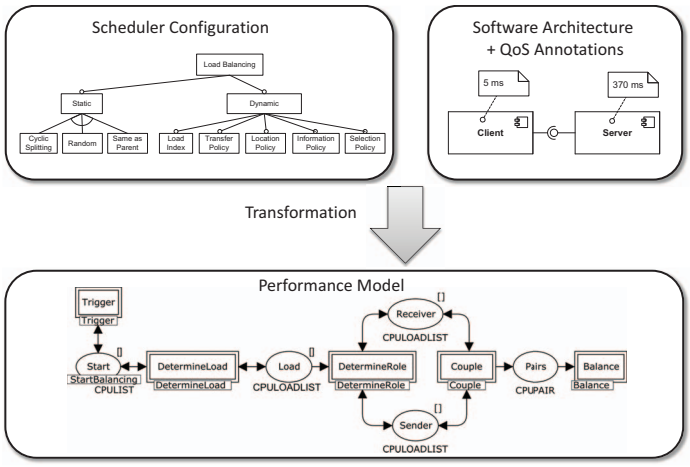


Fig. 1. Overview on application of MOSS.

Figure 1 illustrates the application of MOSS following the idea of model-driven performance prediction [2] where transformations generate performance models based on a set of domain specific languages. In case of MOSS, deployment experts specify the behaviour of the operating system schedulers on an abstract level. The specification includes the load-balancing policy, optimisation strategies for I/O bound process and timesharing strategies for a single processor. Software architects model the components of the system under study, define their behaviour, and estimate resource demands. When completed, a transformation generates analytical performance models (indicated by the timed coloured Petri net in Figure 1) or code for a discrete event simulation [14]. In this paper, we focus on the modelling and analysis of multiprocessor load-balancing policies. First, we present the different load balancing policies and their features that can be used to configure MOSS (Section III). Next, we illustrate how the different configurations can be expressed in terms of timed coloured Petri nets (Section IV). Finally, a case study demonstrates how MOSS increases the prediction accuracy for different operating systems.

III. Features of Multiprocessor Load-Balancing

In this section, we introduce categories of performance-relevant features for SMP load-balancing. We extend the general categorisation of Shivarati et al. [22] (summarized in Figure 2(a)) with concrete characteristics for multiprocessor systems (Figures 2(b) to (f)) based on our previous experiments on performance-relevant influencing factors [7]. We use feature diagrams [5] to model the relevant factors and variation points of MOSS.

In general, GPOS load-balancing policies for SMP environments can be centralised, hierarchical, or fully decentralised. Policies with centralised components suffer from a potential bottleneck and a single point of failure. These limitations affect their scalability and reliability. Hierarchy can reduce these risks, but only fully decentralised systems, where all nodes act independently, can solve these problems [22].

Load-balancing policies can be characterised as static, dynamic, or any combination of these. Figure 2(a) shows relevant features. Static policies use a priori knowledge on the system for balancing decisions. In Figure 2(a), the exclusive choice for static policies offers the features cyclic splitting, same as parent, and random. *Cyclic splitting* assigns tasks to processors in a round robin fashion independently of the task and the processor's load. Similarly, the *random* policy assigns tasks to each processor with a predefined probability. The probability can be equally distributed or varied for different CPUs, e.g, to consider the influence of different processing power. *Same as parent* is specific to multiprocessor environments. It allocates a new task to the same processor as its creator. Thus, it leaves the actual load-balancing to the dynamic policies, which use information on the system state for load-balancing decisions.

The mandatory features of dynamic load-balancing policies in Figure 2(a) determine when and where load-balancing will take place. Each feature is discussed in the following paragraphs.

Load indices estimate the performance of a task on a particular processor (Figure 2(b)). The current *CPU queue length* is known to be the best indicator for a task's performance on a particular node [13]. For multiprocessor systems, various derivations of the CPU queue length can be used, such as the *average CPU queue length* over a predefined *time span* or an *ageing CPU queue length* with a *weight* for the age emphasis.

The *transfer policy* (Figure 2(c)) determines whether a processor should participate in a task transfer as a sender or

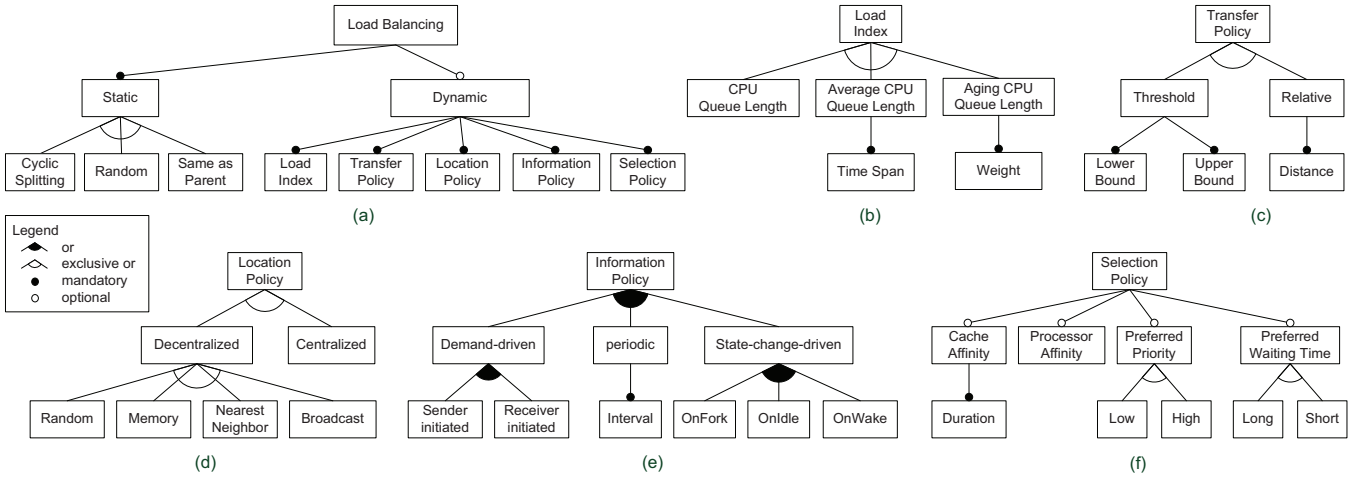


Fig. 2. Feature diagrams for classifying load-balancing policies.

as a receiver. *Threshold*-based policies define an upper and lower bound for a processor's load index. If a processor's load falls below the *lower bound*, it becomes a (potential) receiver. Otherwise, if a processor's load rises above the *upper bound* it becomes a (potential) sender. Between these bounds, the processor does not participate in load-balancing. *Relative* policies consider a processor's load in relation to loads of other processors. Load-balancing is initiated if the load of two processors differs more than a predefined value.

The *location policy* (Figure 2(d)) is responsible for matching senders and receivers. In a centralised system, all information on the processor's state is known and matching is easy. In decentralised systems, the current node cannot know the global system state. So, it can pick a node at *random*, *broadcast* its request to all nodes, choose the *nearest neighbour*, or use information collected during previous calls to find a transfer partner (*memory*). In distributed systems, the different policies vary in chance and overhead for finding appropriate transfer partners. However, they play a minor role for SMP systems.

The *information policy* (Figure 2(e)) determines when information about the states of other processors in the system is collected and load-balancing is triggered. *Demand-driven* policies exchange information whenever a processor becomes a sender (*sender-initiated*) or receiver (*receiver-initiated*). If both cases are possible, the policy is called symmetrically initiated. When collecting data *periodically*, the *interval* determines the period length at which balancing efforts occur. Furthermore, *state-change-driven* policies pass information whenever a node's state changes. The most important events for multiprocessing systems are *OnFork*, which is activated whenever a new task is created, *OnIdle*, which is activated whenever a processor becomes idle, and *OnWake*, which signals that a process requests processing after waiting.

If a transfer is initiated, the *selection policy* (Figure 2(f)) defines which tasks are transferred. In order to reduce negative effects, the selection favours tasks which (presumably) have a long life-span and which have a minimum number of location dependencies. For example, newly originated tasks are preferable for transfer, since they do not have any state that needs to be transferred, there are no cache dependencies and it can be assumed they live relatively long. If the selection policy does not find a suitable task for transfer, it no longer considers the processor as a sender. All selection criteria in Figure 2(f) are optional and can be combined arbitrarily. Selection policies that take into account *cache affinities* migrate only tasks that did not run on the sender for a given *duration*. Avoiding to move tasks with (presumably) cached data allows to reduce overhead. Additionally, *processor affinity* limits the shifting of tasks to a predefined set of processors. When multiple tasks are available for migration, the options *preferred priority* and *preferred waiting time* determine which one to select. If the preferred priority is *high* (*low*), higher (lower) priority tasks are migrated first. Furthermore, if the *preferred waiting time* is *short*, tasks at the end of a queue are preferred over tasks in the beginning of the queue (and vice versa for *long*).

Windows and Linux: The Windows and Linux operating systems differ with respect to multiprocessor load-balancing as we have demonstrated in [7]. While Windows balances as little as possible, Linux aims at keeping the system fairly balanced among the available processors. Windows' static balancing policy uses cyclic splitting to assign newly created tasks to processors. Its dynamic balancing policy realises a threshold-based transfer policy. Windows uses the CPU queue length as a load index. If the load of a CPU drops below one (the CPU becomes idle), the CPU becomes a receiver. All CPUs with a load greater than one are potential senders (threshold-based transfer policy). Once idle, a processor looks for executable tasks on other processors implementing a demand-driven, receiver initiated information policy. Windows' location policy chooses the processor with the highest load as sender. Its selection policy prefers tasks with high priorities, but also considers their processor and cache affinity. The latter directly relates to the time a task last ran. Additionally, processor affinities restrict the selection of processors on which a task can be processed. Windows employs a state-change-driven policy. Whenever a task becomes ready (e.g., after blocking or creation) and an idle processor (receiver) is available, the scheduler tries to migrate the task to the receiver.

In contrast to Windows, Linux uses an ageing CPU queue length as load index. Its relative transfer policy initiates load-balancing only if the distance exceeds a threshold of 2. Furthermore, Linux uses a state-change-driven as well as a periodic information policy. The state-change-driven policy reacts whenever a new task is created (OnFork), a task is about to be awakened (OnWake), or a processor becomes idle (OnIdle). The periodic policy checks at regular intervals if the processors need to be balanced. If the load differs too much, it moves tasks from the busiest processor to the most idle one. The selection policy of the Linux scheduler considers factors like cache affinity and processor affinity. Moreover, it prefers tasks with a low priority and a long waiting time for migration.

IV. Performance Model for Multiprocessor Scheduling

MOSS reflects the influence of load-balancing as well as different time-sharing strategies (run queues, priorities, timeslices) and the preference of I/O-bound and interactive tasks. In the following, we give an overview of the definition of MOSS in terms of timed Coloured Petri Nets (CPNs) [10].

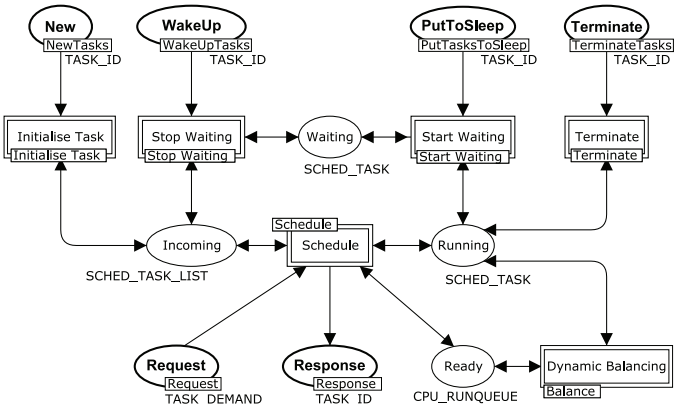


Fig. 3. Overview of the CPN model of MOSS.

MOSS provides a typical GPOS interface in form of interaction points: It accepts requests, i.e., demands of processing time, notifies the calling subnet when one of the requests has been finished, starts processing new tasks, terminates finished tasks, and puts tasks to sleep or wakes them up. Figure 3 gives a schematic overview of the CPN model realising this behaviour. The model's substitution transitions (rectangles with double lines) encapsulate the subnets of the scheduler's time sharing, interactivity, and multiprocessor load-balancing policies as single transitions. The boldly printed places represent interaction points of MOSS to task behaviour models, which require access to scheduled resources. All other places are internal to MOSS. The communication between all subnets is based on fusion places (ellipses with small rectangle attached). Fusion places can be used in multiple subnets. They belong to a fusion set that is given by the name in the rectangle attached to the place. The marking of a fusion place is identical for all places of the set for all subnets.

All places accept tokens of colours printed in Listing 1. MOSS communicates with behavioural models of tasks based on a unique task identifier (`TASK_ID`). For each identifier, the scheduler model manages its internal information (e.g., timeslices and priorities) using colour `SCHED_TASK`.

Listing 1. Basic colour sets of the scheduler model.

```
colset TASK_ID = INT;
colset DEMAND = INT;
colset TASK_DEMAND = (TASK_ID, DEMAND);
colset SCHED_TASK = product ID * CPU_ID
                  * PRIORITY * TIMESLICE timed;
colset SCHED_TASK_LIST = list SCHED_TASK;
colset CPU_RUNQUEUE = product CPU_ID * TASKLIST;
```

When a new task is created, its unique identifier is put on place `New` to notify the scheduler, that a new task requires scheduling. Transition `Initialise Task` then creates the initial scheduling information for the task (`SCHED_TASK`), which contains its initial processor, timeslice, and priority. The transition selects the processor according to the chosen static load-balancing policy (see Figure 2(a)). Finally, it inserts the new token at the end of list `SCHED_TASK_LIST` on place `Incoming`. All tasks have to enter the subnet of substitution transition `Schedule` via this place. Whenever a `SCHED_TASK` is added to this list, transition `Schedule` assigns the task to its processor's run queue. Furthermore, scheduling is only initiated if place `Incoming` is empty. This behaviour ensures that scheduling always considers all tasks ready for execution. Place `Ready` holds a separate run queue (`CPU_RUNQUEUE`) for each processor. It contains those tasks that are ready for execution on that specific processor (or core). Whenever a processor is idle or the currently running task's timeslice expires, transition `Schedule` removes the currently executing task from place `Running` and puts the next executable task of the processor's run queue there.

When a task requests processing time, it puts a `TASK_DEMAND` token on place `Request`. The token contains the task's unique identifier (`TASK_ID`) as well as the demand which needs to be processed (`DEMAND`). As soon as the task is running (i.e., its `SCHED_TASK` token lies on place `Running`), it can reduce its demand according to the time it spent on place `Running`. As soon as a task's demand reaches zero, transition `Schedule` puts its `TASK_ID` on place `Response` to notify the task behavioural model that its request has been processed and that it can continue execution. Transition `Dynamic Balancing` levels the load between multiple processors according to the specified dynamic load-balancing policy.

Furthermore, MOSS reflects the mutual performance influences of passive resources (i.e., semaphores) and schedulers. It may be necessary to put a task to sleep until the resources that have been requested become available. As soon as these resources are available, the scheduler needs to resume processing of the interrupted task. To notify the scheduler about such state changes, a task's unique identifier is put on places `PutToSleep` or `WakeUp`. Transition `Start Waiting` removes the task from the processor it is currently running on and puts its token on place `Waiting`. As soon as a passive resource notifies the scheduler to wake up the

task, transition `Stop Waiting` removes the corresponding token from place `Waiting` and inserts it at the end of the `SCHED_TASK_LIST` on place `Incoming`.

Finally, the behavioural model notifies MOSS of a tasks termination by putting the task's unique identifier on place `Terminate`. Transition `Terminate` then removes the internal `SCHED_TASK` token of that task.

MOSS is refined hierarchically by multiple layers of substitution transitions. The hierarchical structure of CPNs encapsulates the behaviour of all feature configurations in separate subnets. Each substitution transition resembles a possible variation point. The subnets are selected according to a given feature configuration. Each feature may be further subdivided into several smaller parts, which represent its variation points.

Dynamic Load-Balancing: In the scope of this paper, we focus on the subnets for dynamic load-balancing. Additionally, MOSS reflects the influences of timesharing, compute-bound and I/O-bound-load as well as static load-balancing.

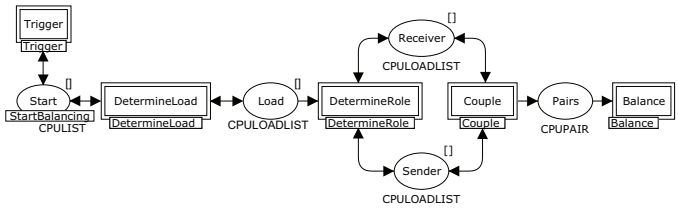


Fig. 4. Overview of dynamic load-balancing.

With respect to dynamic multiprocessor load-balancing, load indices as well as transfer, location, information, and selection policies are modelled in MOSS. We have split the CPN for dynamic load-balancing policies into multiple subnets (see Figure 4), each of which represents one of these policies. Substitution transition `Trigger` activates load-balancing according to the specified *information policy*. When load-balancing has been triggered, transition `DetermineLoad` determines the current *load index* for all processor identifiers on place `Start` and stores the result on place `Load`. Next, transition `DetermineRole` partitions the processors into senders and receivers based on their current load. Whether a processor needs to participate in load-balancing as well as its role depend on the specified *transfer policy*. When all processors have been partitioned and a sender and a receiver are available, transition `Couple` creates pairs of potential senders and receivers. Transition `Balance` models the movement of tasks from one processor to another. It chooses the tasks for transfer according to the defined *selection policy*. In the following, we present the realisation of substitution transitions `DetermineLoad`, `DetermineRole`, `Couple`, and `Balance`.

Determining the Load: When load-balancing has been initiated, then the load of each processor has to be determined in the next step. Figure 5 depicts the subnet for the computation of the current *CPU queue length*. It collects the necessary information from places `Ready` and `Running` and stores the resulting load in the list on place `Start`. For this purpose, transition `DetermineCurrentLoad` is enabled as soon as an element is added to the `CPU_LIST` on place `Start`. Furthermore, a bidirectional arc with an empty list ensures that the `TASK_LIST` on place `Incoming` is empty (inhibitor arc pattern [18]).

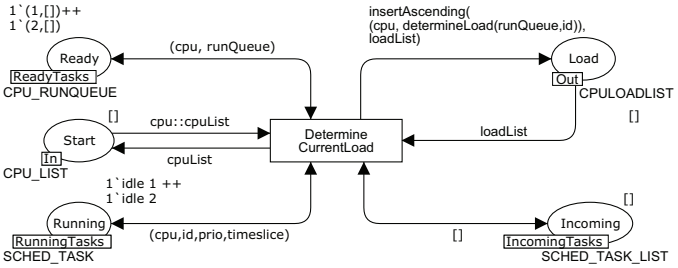


Fig. 5. Subnet determining the current processor load (CPU Queue Length).

When firing, transition `DetermineCurrentLoad` removes the first element from the `CPU_LIST` on place `Start` and gets the corresponding run queue (from place `Ready`) as well as the currently running task (from place `Running`). Furthermore, it adds a new token of colour `CPULOAD` (cf. Listing 2) to the list on place `Load`. The `CPULOAD` embodies a `CPU` representing the processor's identifier and an integer representing its load. Function `insertAscending` (cf. Listing 2) realises the priority queue pattern [18] and ensures that processors on place `Load` are ordered according to their current load. Finally, transition `Determine Current Load` uses function `determineLoad` to compute the processor's load from the run queue and the executing task's identifier (cf. Listing 2). In addition to the CPU queue length, MOSS provides a subnet for age based load indices.

Listing 2. Functions `determineLoad` and `insertAscending`.

```
colset CPULOAD = product CPU * INT;
colset CPULOADLIST = list CPULOAD;

fun determineLoad (runQueue, id) =
   if id = IDLE_ID then length runQueue
                   else length runQueue + 1;

fun lowerLoad ((cpu1,load1), (cpu2,load2)) = (load1 < load2);

fun insertAscending (elm,[]) = [elm]
    | insertAscending (elm,(q::queue)) =
        if lowerLoad (elm,q)
           then elm::q::queue
           else q::(insertAscending(elm, queue));
```

Determine Senders and Receivers: After the load of all processors has been determined, senders and receivers for balancing need to be identified. The *transfer policy* determines how the scheduler classifies the processors based on their current load index. For the *threshold-based* policy depicted in Figure 6, one of the transitions `IsReceiver`, `IsBalanced`, or `IsSender` fires depending on the processor's load.

If the load of a processor lies below the `lowerBound`, transition `IsReceiver` considers the processor as a receiver and inserts its identifier into the list on place `Receiver`. For the insertion, transition `IsReceiver` calls function `insertAscending`. Analogously, transition `IsSender` fires if a processor's load lies above the `upperBound`. Function `insertDescending` adds the tuple `(cpu, load)`

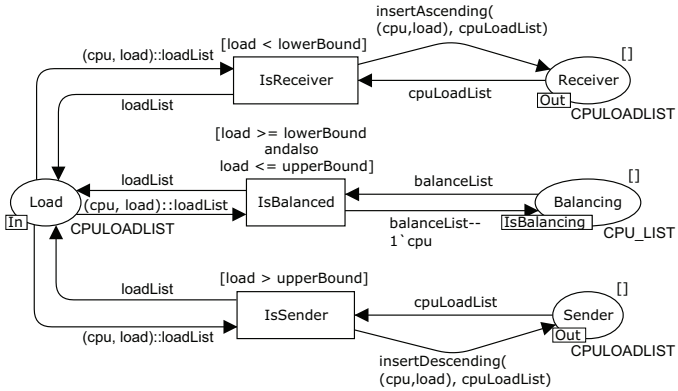


Fig. 6. Subnet for the threshold-based transfer policy.

to the list on place `Sender` in descending order. By sorting the receivers in an ascending order and the senders in a descending order, MOSS can directly identify the highest loaded and least loaded processors. Finally, transition `IsBalanced` fires if no balancing for the selected processor is necessary. The transition simply removes the processor from the list of processors and, thus, aborts balancing for this processor.

Finding Partners for Transfer: To create a balanced situation for all processors of a system, the load-balancing policy needs to identify transfer partners, i.e., senders and receivers, so that tasks can be moved from one to the other. The subnet of substitution transition `Couple` realises the identification of fitting transfer partners in the context of MOSS.

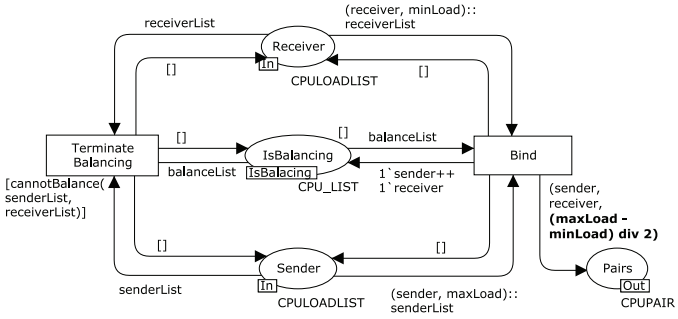


Fig. 7. Subnet for substitution transition `Couple`.

Figure 7 depicts its subnet with the input places `Sender` and `Receiver` and the output place `Pairs`. If this subnet, either transition `TerminateBalancing` or `Bind` is enabled. The first ensures that the balancing attempt is terminated if no redistribution of load is possible. Its guard calls function `cannotBalance` which checks whether the `receiverList` or whether the `senderList` is empty while its counterpart still contains at least one element. In this case, the system is either overutilised (contains only senders) or underutilised (contains only receivers) and balancing is not possible. Thus, transition `TerminateBalancing` removes all senders and receivers, and empties the list of currently balanced tasks on place `IsBalancing`.

If, otherwise, the lists on places `Sender` and `Receiver` contain at least one element each, transition `Bind` is enabled.

Listing 3. Function `cannotBalance`.

```
colset CPUPAIR = product CPU * CPU * INT;

fun cannotBalance (senderList, receiverList) =
    (length receiverList = 0 andalso length senderList > 0)
    or else
    (length senderList = 0 andalso length receiverList > 0)
```

It takes the first sender and receiver token from the lists on places `Sender` and `Receiver` and puts a new `CPUPAIR` token on place `Pairs`. The token contains the sending and receiving processor's identifiers as well as the number of tasks to move. By default, MOSS assumes that threshold based policies just move a single task while relative policies equalise the load of the sender and receiver. Furthermore, transition `Bind` terminates the load-balancing attempts for all other processors and removes their tokens from places `Sender`, `Receiver`, and `IsBalancing`. The termination is necessary, since the load-balancing of the two selected processors changes the overall load distribution. If further balancing is required, a whole new balancing attempt must be started to determine the new senders and receivers. For example, one of the processors involved in the current load-balancing attempt may still be the busiest processor after balancing is finished. Continuing load-balancing with the remaining processors would not resolve such situations.

Balancing the Load: If two transfer partners have been identified, transition `Balance` (Figure 8) can select and move tasks from the sender to the receiver. For each pair on place `Pairs`, it moves the specified number of tasks from the sending to the receiving processor. The task transfer is executed in multiple steps. During each step, transition `MoveTask` moves one task from the sender to the receiver until no further tasks have to be moved. Then transition `BalancingFinished` terminates the load-balancing operation.

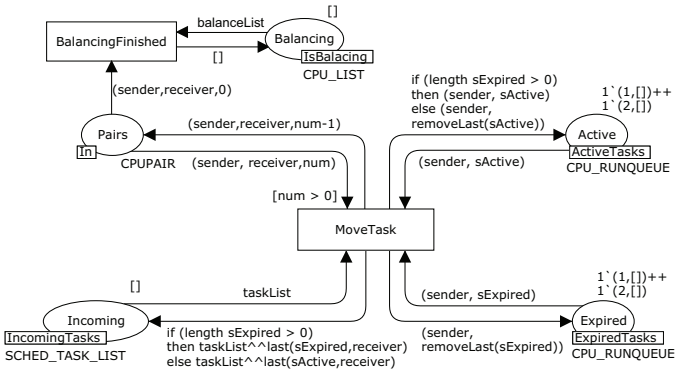


Fig. 8. Subnet for substitution transition `Balance`.

Transition `MoveTask` takes a `CPUPAIR` token from place `Pairs`, whose number of tasks to move is greater than zero (`num > 0`). Furthermore, it selects the sender's (active and expired) run queue from places `Active` and `Expired`. Transition `MoveTask` first tries to move the last task of the expired queue. If such a task exists (i.e., `length sExpired > 0`), transition `MoveTask` removes it from the sender's expired run queue and adds it to the list of tasks on place incoming

(`taskList^^last(sExpired, receiver)`). If the expired run queue of the sender is empty, transition `MoveTask` selects a task from the active run queue instead. Function `last` returns the last element of a run queue and sets its processor identifier to the specified one (cf. Listing 4). Using functions `last` and `removeLast` realises a *selection policy* where the preferred priority is low and where the preferred waiting time is short. Processor affinities as well as cache affinities are not considered.

Listing 4. Functions `last` and `removeLast`.

```
fun last ([], newCpu) = []
  | last ([(cpu,id,prio,timeslice)], newCpu)
                = [(newCpu, id, prio, timeslice)]
  | last (q::queue, newCpu)= last(queue, newCpu);

fun removeLast [] = [] | removeLast [elm] = []
  | removeLast (q::queue) = q::removeLast(queue);
```

Transition `MoveTask` puts the token of the task moved on place `Incoming`. The scheduler's subnet automatically adds the token to the correct run queue of the receiving processor. Furthermore, transition `MoveTask` returns the `CPUPAIR` token to place `Pairs` and reduces its number of tasks by one. If the number is still greater than zero, transition `MoveTask` is enabled again to move the next task. As soon as the number of tasks to move reaches zero, transition `BalancingFinished` is activated. It empties the list of currently balanced processors on place `Balancing`. This terminates the balancing operation.

V. Case Study

The case study presented in this section demonstrates the applicability and prediction accuracy of MOSS. For this purpose, we focus on the influence of different workloads and of different operating systems schedulers on performance. We minimise the impact of other components and services that are typically used in the chosen application scenario. Simplification is necessary to avoid disturbances of other system components and focus solely on the effect of scheduling. This approach achieves a high internal validity of the results at the cost of a low external validity. The case study is placed in the scenario of a supply chain management for supermarkets (as described in [23]). In the following, we evaluate the performance of business intelligence reporting, online monitoring, and requests to static web pages.

TABLE I
Request types used in the case study.

Request Type	Mean Service Time	Relative Frequency
Web Page	5 ms	70%
Monitoring	250 ms	20%
Reporting	3000 ms	10%

From observations of the current system, we expect a strongly fluctuating load with burst periods of 5 to 10 minutes. To reflect this behaviour, the workload is modelled with a sinus function and a period of 20 minutes. Its low periods represent the system's usual workload, its high periods represent the burst conditions. The rate fluctuates between a factor of $1/3$ and 1 of the specified rate (requests per minute, req/min). In our experiments, we used open workloads and maximal arrival rates of 180 req/min, 360 req/min, and 720 req/min. These rates result in a full utilisation of a system with one, two, and four cores, respectively. This workload allows estimating the influence of burst periods on system's response times, resource utilisation and throughput. For sake of simplicity, we assign a fixed and deterministic mean service time and relative frequency to each request type as listed in Table I.

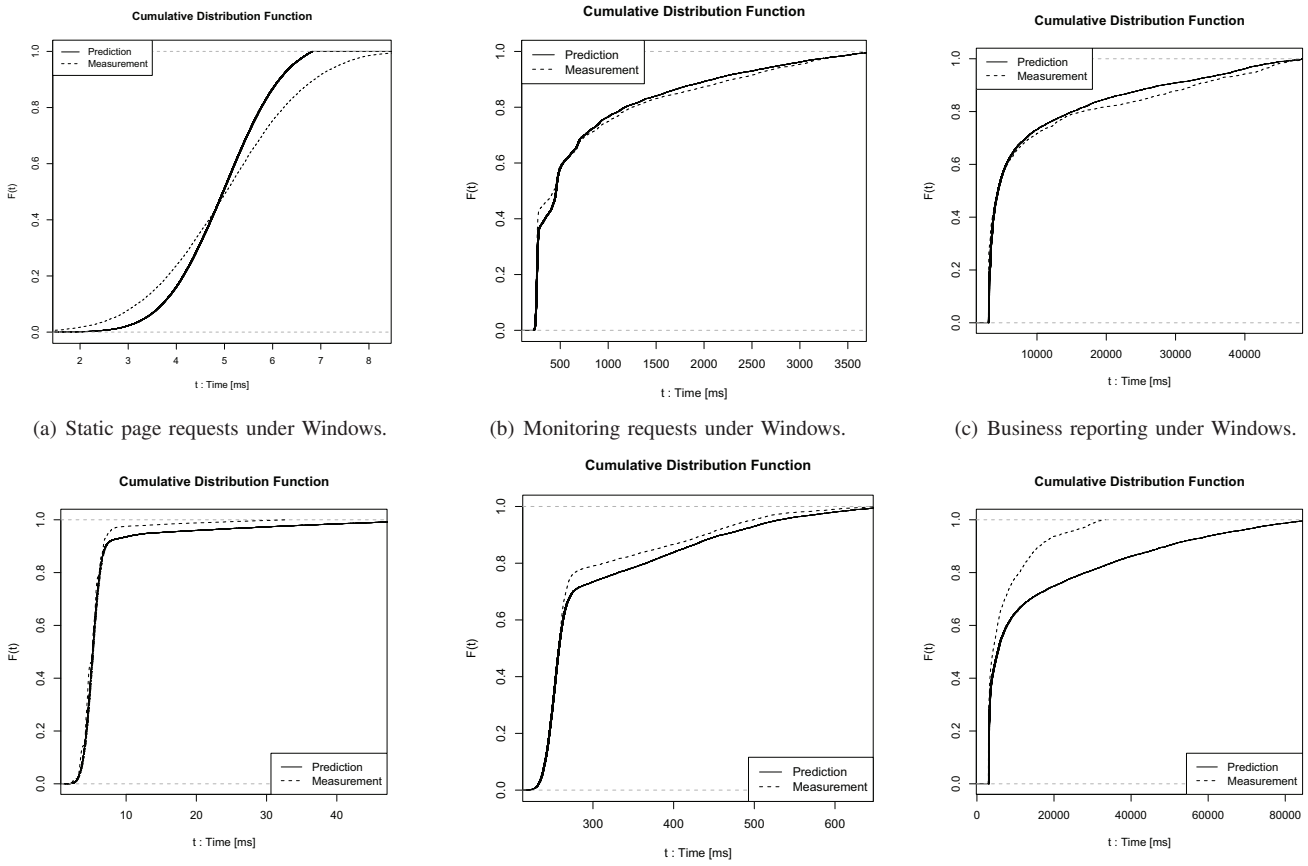
For the case study, we implemented the application in Java using the prototype framework [3] and instrumented it for measurements. The implementation ensures that the case study is focussed on scheduling effects and excludes disturbances of the environment. All measurements were taken on a single machine with the accuracy of the machine's clock frequency. For performance prediction, a discrete event simulation built on SSJ [14] and specialised for MOSS has been implemented and integrated with the Palladio Component Model (PCM) [3].

Results: Figure 9 shows the cumulative distribution functions (cdfs) of the response times for page requests, online monitoring, and business reporting for Windows Server 2003 and Linux 2.6.22 on a dual-core system (AMD Athlon 64 X2 5200+, 2.61 GHz, 2 GB RAM).

Figures 9(a) and 9(d) show the cdfs predicted and measured for the response time of page requests (service time of 5 ms) under Windows Server 2003 and Linux 2.6.22, respectively. For Windows Server 2003, the predictions comply to the measurements even though the measurements have a slightly higher variance. The predictions and measurements for Linux 2.6.22 illustrate the role of outliers in the results of this case study. Approximately 95% of all requests are processed within 5 ms for predictions and measurements. However, processing of the upper 5% of all requests is delayed up to several seconds.

Figures 9(b) and 9(e) show cdfs of the response times for online monitoring (service time of 250 ms). The predictions and measurements widely overlap for the Linux and Windows operating system. Furthermore, the predicted and measured mean values and medians deviate no more than 15%. The response times for monitoring requests differ significantly for both operating systems. Linux limits the distribution from 250 ms to 650 ms, while the response time under Windows ranges from 250 ms to 3600 ms. The response time's median for monitoring requests under Linux (259 ms) is approximately half of the one under Windows (459 ms).

The different response times observed for both operating systems are a consequence of the dynamic prioritisation of tasks. Linux' slowly decaying dynamic priorities ensure that requests to the online monitoring are only interrupted by requests that received a similar amount of processing time (assuming they started at similar priorities). Therefore, online monitoring preempts all reporting requests that received more than 250 ms of processing time. By contrast, Windows grants only a small priority bonus to all incoming tasks. The boost is independent of their processing time and lasts approximately



(a) Static page requests under Windows.

(b) Monitoring requests under Windows.

(c) Business reporting under Windows.

(d) Static page requests under Linux.

(e) Monitoring requests under Linux.

(f) Business reporting under Linux.

Fig. 9. Comparison of prediction and measurements for a dual-core system with Windows Server 2003 and Linux 2.6.22.

60 ms. While short requests (smaller than 60 ms) benefit from this policy, longer requests, which cannot be completed within this period, may be delayed. Thus, requests to online monitoring (that last approx. 250 ms) compete with the business intelligence reporting.

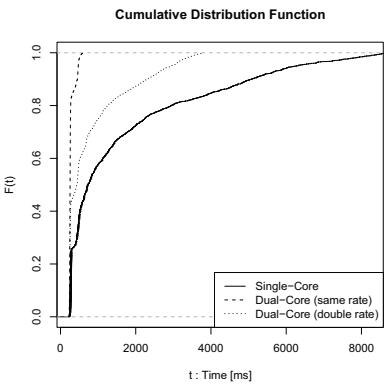
The median of measured response times for the business reporting (service time of 3 s) are comparable for both operating systems (Windows: 4.6 s, Linux: 4.4 s). However, the predicted and measured response times deviate by 30% for the business reporting under Linux (Measured: 4.4 s, Predicted: 5.7 s). This is the largest deviation of predictions and measurements that we observed in our case study. One cause is the high load of the system. Under Linux, the beginning of requests during peak periods was deferred which lead to the observed deviation.

The Scheduler's Influence on Scalability: In this section, we evaluate the influence of the operating system scheduler on software performance for i) different numbers of processor cores (one, two, and four) and ii) different load intensities (180 req/min, 360 req/min, and 720 req/min). The load intensities and the number of cores are doubled in each step, so that the load per core stays similar.

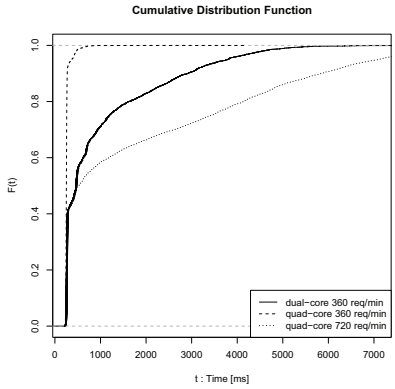
Figure 10(a) compares the response times for a singe-core and dual-core system for a workload of 180 req/min and 360 req/min for requests with a service time of 250 ms (online monitoring). Under Windows, the median of the response time decreases from 814 ms to 256 ms (see Table 10(c)). By contrast, the additional core does not affect the response time's median under Linux, but reduces its mean value (from 1217 ms to 284 ms). This reduction is caused by the vanishing of the heavy tail of the distribution.

If the server's load doubles from 180 req/min to 360 req/min, response times still improve by a factor up to two. All requests benefit from the second core even though the load per processor core is the same as for the single-core scenario. The speedup of the dual-core processor is a consequence of the multiprocessor load-balancing. For dual-core processors, the operating system scheduler moves tasks as soon as one of the processors becomes idle (Windows) or is less loaded (Linux). Load-balancing between the processor cores reduces the delay for each request. If tasks are moved from one core, the remaining tasks can be processed with shorter waiting times. The tasks moved to another core find a less contended processor and, thus, receive a larger share of processing time. As a result, a major decrease of the response times can be observed, especially under Windows.

The situation changes when we move from a dual-core processor to a quad-core processor (Figure 10(b)). For a workload of 360 req/min, the additional processor cores still provide a significant improvement with respect to response times.



(a) Single-core vs. dual-core (Windows).



(b) Dual-core vs. quad-core (Windows).

#cores	Operating System	req/min 180	360	720
single	Linux	261	-	-
	Windows	814	-	-
dual	Linux	255	258	-
	Windows	256	458	-
quad	Linux	305	305	286
	Windows	250	252	523

(c) Response time median (ms).

Fig. 10. Scheduler influence on response times for different numbers of processor cores and different workload intensities.

However, when the system's load doubles to 720 req/min the response time of the quad-core system is larger than the response time of the dual-core system with 360 req/min. Again, the load per processor core is the same. By contrast, the median is stable under Linux and no performance loss can be observed for a load of 720 req/min. The difference is caused by the different load-balancing strategies. Windows cannot sufficiently balance the load among all four processor cores, since it initiates balancing only if a processor becomes idle. By contrast, Linux' active balancing policy can distribute the load efficiently and maintain a higher performance. However, all measured response times were higher for the quad-core system. This observation can be a consequence of the active balancing that leads to many processor switches and balancing attempts in scenarios with many small computational bursts [11].

Decision Support Example: In our setting, we want to identify the appropriate operating system (Linux 2.6 or Windows Server 2003) and number of cores that provide the most stable performance. We are mainly interested in the mean response times for different requests (business reporting, monitoring, and static page).

With the previous results, we can state that a single- or dual-core processor can handle the application's workload only insufficiently. Especially during peak load, the response time increases by several orders of magnitude. Windows poses a significant delay on the online monitoring (up to 6 s) which is not acceptable. Using Linux, such heavy load can lead to timeouts for requests to static pages in more than 5% of all requests. However, we accept this trade-off. The best alternative is to deploy the application on a system with a quad-core processor running Linux 2.6. The major performance gain by the two additional cores provides acceptable responsiveness in heavy load conditions.

VI. Related Work

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

General Purpose Operating Systems: Kluge et al. [11] 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 an MPI application in three different scenarios. The results of Kluge et al. pointed out strong mutual dependencies between multi-processor 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. Our approach takes these factors into account.

In [4], Bulpin and Pratt evaluate the performance of different SPEC CPU2000 benchmarks in a 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%. MOSS is not focussing on SMT technologies, but on performance properties of GPOS schedulers running in a non-SMT environment.

MOSS provides a GPOS model for performance analysis. To the best of our knowledge, no other approach is available so far that provides a detailed GPOS scheduling model for the performance analysis of distributed systems, such as business information systems. Approaches like [26] use LQN models for performance analysis, but include only simplified scheduling models such as FIFO or processor sharing.

Single Processor and Multi Processor Scheduling in Queueing Theory: While scheduling policies for single-server systems (extended policies described in [17], [25], [1]) are well studied and analytically tractable, multi-server queueing models pose several new challenges [24]. 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 [16]. An optimal strategy for multi-server systems is yet unknown.

For multi-servers systems (including multi-core processors), various load distribution techniques have been analysed [9], [20], [19]. However, these solutions have a limited availability, i.e., they address specific combinations of scheduling and routing policies. For example, Harchol-Balter et al. [9] analysed multi-server systems with prioritisation. They came to the conclusion that the effects of prioritisation in multi-server systems cannot be predicted by considering a comparable single-server system. Thus, prediction models for GPOS schedulers have to be enhanced regarding multi-server scheduling policies. We designed MOSS in order to take multi-server scheduling as well as prioritisation into account.

VII. Conclusions

In this paper, we introduced MOSS, a performance model for general purpose operating system schedulers. MOSS accurately predicts the effect of different scheduler configurations and load balancing strategies on software performance. The integration of MOSS with the Palladio Component Model enables design-time performance predictions of component-based software applications considering the influence of different operating system schedulers. Performance analysts and software architects can directly benefit from the increased prediction accuracy provided by MOSS. To allow a flexible and realistic customisation of MOSS, we classified performance-relevant features of operating system schedulers. The classification is based on the properties and the effects of each feature that we have identified in earlier experiments [7]. Finally, we evaluated the prediction accuracy of MOSS in a larger case study. We were able to demonstrate that the prediction error was below 5% to 10% in most cases. Only for long business reporting requests under Linux the measurements and predictions deviated about 30%.

Furthermore, the results demonstrated how MOSS can support decision making. MOSS can help software architects and performance analysts to estimate the performance (response time, throughput, and resource utilisation) of software applications in SMP environments. It supports design-time as well as deployment decisions. Software architects and performance analysts can reliably assess the performance gain of concurrency in their application for different numbers of processors and cores as well as different operating systems. Since concurrency significantly increases the complexity of a software system [15], MOSS also aids in deciding for the appropriate degree of concurrency.

In this paper, MOSS has been evaluated for Linux 2.6.22 and Windows Server 2003 and accurately represents their behaviour. As a next step, we want to capture additional scheduler features and support a broader range of operating systems. Furthermore, we strive for an efficient mechanism to extend and adjust MOSS for newly evolving operating system schedulers. For this purpose, we are developing a technique for automatically deriving the best configuration for MOSS based on systematic measurements on a target platform. Such an approach could help creating configurations for other operating systems.

References

[1] S. Aalto, U. Ayesta, S. Borst, V. Misra, and R. N. nez Queija. Beyond Processor Sharing. *ACM SIGMETRICS Performance Evaluation Review*, 34(4):36–43, 2007.

[2] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *Transactions on Software Engineering*, 30(5):295–310, May 2004.

[3] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *JSS*, 82:3–22, 2009.

[4] J. Bulpin and I. Pratt. Multiprogramming Performance of the Pentium 4 with Hyper-Threading. In *Proceedings of WDDD 2004*, pages 53–62, 2004.

[5] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[6] G. Franks, P. Maly, M. Woodside, D. Petriu, and A. Hubbard. Layered Queueing Network Solver and Simulator User Manual, May 2007.

[7] J. Happe, H. Groenda, and R. H. Reussner. Performance Evaluation of Scheduling Policies in Symmetric Multiprocessing Environments. In *Proceedings of MASCOTS'09*, pages 1–10, Sept. 2009.

[8] J. Happe, H. Koziolek, and R. H. Reussner. Parametric Performance Contracts for Software Components with Concurrent Behaviour. In F. S. de Boer and V. Mencl, editors, *Proceedings of FACS 2006*, volume 182 of *ENTCS*, pages 91–106, 2006.

[9] M. Harchol-Balter, T. Osogami, A. Scheller-Wolf, and A. Wierman. Multi-Server Queueing Systems with Multiple Priority Classes. *Queueing Systems: Theory and Applications*, 51(3-4):331–360, 2005.

[10] K. Jensen, L. M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *STTT Journal*, 9(3):213–254, 2007.

[11] M. Kluge and W. E. Nagel. Analysis of Linux Scheduling with VAMPIR. In *Proceedings of ICCS 2007*, volume 4488 of *LNCS*, pages 823–830. Springer-Verlag Berlin Heidelberg, 2007.

[12] S. Kounev. Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets. *Transactions on Software Engineering*, 32(7):486–502, July 2006.

[13] T. Kunz. The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. *TSE*, 17(7):725–730, 1991.

[14] P. L'Ecuyer, L. Meliani, and J. Vaucher. SSJ: A Framework for Stochastic Simulation in Java. In *Proceedings of WCS 2002*, pages 234–242. IEEE CS, 2002.

[15] E. A. Lee. The Problem with Threads. *IEEE Computer*, 39(5):33–42, May 2006.

[16] S. Leonardi and D. Raz. Approximating Total Flow Time on Parallel Machines. In *STOC '97*, pages 110–119. ACM, New York, NY, USA, 1997.

[17] S. T. Leutenegger and M. K. Vernon. The performance of multiprogrammed multiprocessor scheduling algorithms. In *SIGMETRICS '90*, pages 226–236. ACM, 1990.

[18] N. Mulyar and W. van der Aalst. Patterns in Colored Petri Nets. Technical report, Eindhoven University of Technology, Department of Technology Management, Apr. 2005.

[19] T. Osogami. *Analysis of multi-server systems via dimensionality reduction of Markov chains*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2005.

[20] B. Schroeder and M. Harchol-Balter. Evaluation of Task Assignment Policies for Supercomputing Servers: The Case for Load Unbalancing and Fairness. *Cluster Computing*, 7(2):151–161, 2004.

[21] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: a cautionary tale. In *Proceedings of NSDI'06*, pages 239–252, Berkeley, CA, USA, 2006. USENIX Association.

[22] N. G. Shivaratri, P. Krueger, and M. Singhal. Load Distributing for Locally Distributed Systems. *IEEE Computer*, 25(12):33–44, 1992.

[23] SPEC. SPECjms2007 Benchmark. http://www.spec.org/jms2007/.

[24] M. S. Squillante. Stochastic analysis of multiserver systems. *SIGMETRICS Perfomance Evaluation Review*, 34(4):44–51, 2007.

[25] A. Wierman and M. Harchol-Balter. Classifying scheduling policies with respect to unfairness in an M/GI/1. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):238–249, 2003.

[26] X. Wu and M. Woodside. Performance modeling from software components. *SIGSOFT Software Engineering Notes*, 29(1):290–301, 2004.