Deadline Prediction Scheduling based on Benefits

Javier Palanca, Marti Navarro, Ana García-Fornes and Vicente Julian

DSIC - Universitat Politècnica de València
Camino de Vera s/n, 46022
Valencia, SPAIN

jpalanca@dsic.upv.es

Abstract

This paper describes a scheduling algorithm that composes a scheduling plan which is able to predict the completion time of the arriving tasks. This is done by performing CPU booking. This prediction is used to establish a temporal commitment with the client that invokes the execution of the task. This kind of scheduler is very useful in scenarios where Service-Oriented Computing is deployed and the execution time is used as a parameter for QoS. This scheduler is part of an architecture that is based on the Distributed Goal-Oriented Computing paradigm, which allows agents to express their own goals and to reach them by means of service compositions. Moreover, the scheduler is also able to prioritize those tasks which provide greater benefits to the OS. In this work, the scheduler has been designed in several iterations and tested by means of a set of experiments that compare the scheduler algorithm with a representative set of scheduling algorithms.

Keywords: Guaranteed processor, Service-Oriented Computing, Scheduling

1. Introduction

Nowadays, service-oriented applications are a new way of developing flexible and distributed solutions. In applications of this kind, the Quality of Service (QoS) has become one of the highest priorities for service providers and their clients. Due to the dynamic and unpredictable nature of the web, providing an acceptable QoS is quite a challenging task. Service consumers need guarantees that the services will be executed with a minimum level of quality. For this reason, service providers must offer and fulfill this commitment to quality.

Service execution time is one of the most important QoS parameters in web services. There are some systems, such as the RT-MOVICAB-IDS system [Herrero et al. (2011)], where the execution of a service on time is considered to be a critical parameter. Therefore, it is necessary to give a valid response before a certain instant in time. Otherwise, the system may become inefficient.

Time as a QoS parameter is not only important when executing a single service but also when a set of services is executed as part of a service composition. In this case, the problem resolution requires the collaboration of several services and, therefore, both the execution time of each individual service and the execution time of the service composition that fulfills the goal must be taken into account.

There are several proposals that introduce the time parameter into web services as a form of expressing temporal constraints. Some of these proposals are shown in [Pan (2005)], [Hao and Zhi-jian (2006)], and [Martin-Diaz et al. (2005)]. Moreover, there are other proposals such as [Naseri and Towhidi (2007)], [Solanki et al. (2004)], and [Fernández-Olivares et al. (2007)] that use the time parameter to guide the service composition. In all these proposals, the time parameter is considered from the point of view of the descriptive level, but at the execution level it provides no guarantee that these time constraints will be fulfilled. Moreover, the service execution time is often not known. This is because this time parameter may depend on many factors that are not controlled by the service provider, such as the system workload where the service is executed or the ending of other services that were previously necessary to execute the service. In these cases, the system must be able to diagnose how long it takes to complete each service. This is a very hard and complex process to procure with current architectures, since they are not focused on time predictability.

Therefore, it is very important to be able to determine the completion time of a single service or a service composition taking into account how the system workload is and, based on this, the system must be able to predict when the service or the composition ends. With this information, the system could become more efficient and could establish a commitment with the service client indicating when it is going to fulfill the service goal. When the system executes a service on time, it may gain a benefit offered by the service client. Moreover, the more quickly the services are fulfilled, the greater benefit the system gains. Thus, it would be desirable to have mechanisms that help to fulfill all services on time with the highest possible benefit.

In previous works, we presented an operating system (OS) architecture [Palanca et al. (2012)] that has the ability to control the service execution and also the ability to build service compositions taking into account the temporal constraints that the services have. This proposal increased the abstraction level provided by the operating system and their services. This allows us to offer an OS execution layer that is integrated into the network and also to offer security and reliability mechanisms, which are not available in lower levels of the OS architectures. Our OS architecture is based on a new paradigm called *Distributed Goal-Oriented Computing* [Palanca et al. (2011)]. This approach is based on Service-Oriented Computing concepts. Its purpose is to find solutions to problems through composition and execution of various services offered by different agents, taking the goal to be achieved as a starting point. This Distributed Goal-Oriented Computing paradigm suggests that agents are the components that provide services in a ubiquitous environment where users express their goals. Thus, users can reach a solution by finding a plan that achieves the selected goal with little user interaction.

The OS needs a scheduler to establish a proper commitment that guarantees that services end at a time agreed upon by both the client and the provider. This module is an important part of the OS since it is responsible for distributing the CPU time among all of the services in execution. The scheduler distributes the CPU time by means of a scheduling algorithm. To do this, the scheduling algorithm plans the order of the execution of the services that were invoked by the distributed environment. There are many schedulers used for general purpose operating systems. These approaches, as discussed below, do not satisfy all the needs of the Distributed Goal-Oriented Computing paradigm. Thus, in this paper, we propose new scheduling algorithms that are not only able to schedule tasks from a distributed environment [Hsu et al. (2011)], but they are also able to analyze when the service ends its execution and to plan the delivery of services with the intention of maximizing the benefit obtained by the OS. In this paper, we also compare the proposed scheduling algorithms with other classical scheduling algorithms.

The paper is organized into the following sections: Section 2 describes the state of the art for scheduling algorithms. Section 3 presents a new scheduling algorithm based on planning-based scheduling. Section 4 presents a set of experiments to compare and refine the different algorithms presented in this paper. Finally, Section 5 presents our conclusions.

2. Scheduling algorithms

As started above, this paper presents a new scheduling algorithm that allows the OS to make a scheduling plan that takes into account the prediction of when a task is going to finish its execution and how much benefit (in quantitative terms) this execution will bring to the OS. This section explores a set of representative schedulers that use different techniques to share the processor (fair algorithms, real-time algorithms, or non-preemptive algorithms). In this section, we also explore a category of schedulers called Planning-based scheduling which are closely related to the scheduler presented in this work.

General purpose Operating Systems schedulers usually use fair algorithms that equally share the processor time among all of the tasks. The most representative example is the Round Robin scheduling algorithm. This algorithm divides the processor time into units called *quantums* and gives each task the same number of quantums. When a task is running, it consumes its quantums; when it runs out of time quantums, the task is expelled from the processor in order to let another task with remaining quantums enter. This algorithm shares the same amount of time with all the tasks, so the feeling of interactivity is very high. However, this kind of scheduler algorithm does not allow the time when a task ends to be predicted because it is not possible to predict how many tasks the processor resources must be shared with.

There has been an interesting evolution of scheduling algorithms that try to share the processor time fairly. There is an algorithm called the *Completely Fair Scheduler* (CFS) by [Wong et al. (2008)]. This algorithm was designed as a replacement of the $O(1)$ algorithm in the Linux kernel. Instead of using queues, this algorithm uses a complex structure called red-black trees. Red-black trees have a time complexity of $O(log_n)$ for insert, search, and delete operations. From the red black tree, CFS efficiently picks the process that has used the least amount of time (this process is stored in the leftmost node of the tree). Even through this is a very fair scheduling algorithm that boosts multi-tasking performance, it is unable to predict when the tasks are going to finish.

There exists a very simple algorithm called FCFS (First-Come First-Served), which serves the tasks in order of arrival. This algorithm was very common in batch systems and implemented a queue that held the tasks in the order they came in. It is a non-preemptive algorithm, which means that until the active task is not finished it will not be interrupted. Of course, with this kind of algorithm, it is easy to make a prediction of the deadline because, in the worst case, it will be its WCET[1], since it is certain that the task will not be interrupted. Its main drawback is that it is a non-preemptive algorithm. In a world where multi-tasking and interactivity are not just features but are requirements, this is not acceptable for distributed systems, which is a focus.

However, predicting the deadline of a task is also a different problem than those found in real-time problems. Real-time systems have to schedule tasks that must be run before an established instant of time, which is also called *deadline*. This deadline is mandatory in hard real-time systems, although deadline fails are not critical in soft real-time systems. These real-time systems use specific schedulers that ensure that every accepted task will be run before its deadline. For this reason, it is very important to have an accurately calculated WCET.

The most simple way of scheduling real-time tasks is to use a *cyclic executive*. This executive is a way of scheduling tasks in a real-time system that using cyclic tasks. These tasks have a period and a WCET, so the cyclic executive just has to create a fixed execution plan provided by the system designer. The cyclic executive can replace the whole OS since it only takes an infinite loop to run the tasks with the order established by the designer.

Planning-based Scheduling is a kind of real-time dynamic scheduling that gives assurances to arriving jobs by implementing admission controls. These assurances are related to the ability of the system to meet the time constraints (deadlines) of the incoming tasks.

Planning-based scheduling (PBS) usually involves making a plan to run all the enqueued tasks, which implies assigning priorities to the tasks. When dynamic priorities are used, the relative priorities of tasks can change as time progresses and also when tasks are executed.

In general, PBS has to go through three steps: Feasibility analysis, Schedule construction, and Dispatching. The feasibility analysis is done to check the schedulability of a task (i.e., whether or not the time constraints of the task can be satisfied). This feasibility test is usually done when the task arrives to the system. These tests are more suited for periodic tasks since they have a periodic activation and the resources they need can be easily calculated and reserved. In planning-based approaches, this test is also applied

[1]Worst-Case Execution Time: the maximum length of time a task could take to execute on a specific hardware platform.[Lv et al. (2009)]

to aperiodic tasks.

The schedule construction is the process of ordering the tasks to be executed. This order is stored for use in the dispatch step. This schedule construction is usually done when dynamic priorities are assigned. Finally, the dispatch step is in charge of deciding which tasks to execute next. This dispatch process may be to follow the established plan, depending on whether the system is preemptive or non-preemptive, the nature of the execution platform, or if the schedule construction is done as part of the feasibility analysis.

There are two reference algorithms for Planning-based Scheduling: RED and Spring. RED (Robust Earliest Deadline) [Buttazzo and Stankovic (1993)] is a robust scheduling algorithm that deals with aperiodic tasks in overloaded environments. Robust schedulers separate timing constraints and importance by considering two different policies: one for task acceptance and one for task rejection. Spring is the algorithm implemented in the Spring Kernel that has a planning-based admission control algorithm that can also accommodate resource requirements of tasks beyond CPU resources.

3. Deadline Prediction Scheduler

One of the most important values that defines the quality of a service provider is the accuracy of the temporal commitment that has been established. As stated above, the client agents that are requesting a service execution will take into account different values that set the value of quality. One of these values is *time*. Client agents can use the deadline of a service to assess how good that service is. While every agent is free to use its own selection filter, the most commonly used filter is to choose those services with the lowest deadline. That is why service provider agents will try to offer the lowest deadline, but always being careful to meet the temporal commitments.

In this section, we are going to present the Deadline Prediction Scheduler (DPS). In order to make a clear presentation of the DPS, we will present different iterations of the algorithm that include different improvements at each refinement.

3.1. The problem of scheduling with deadline predictions

The WCET value provides the maximum time that a service requires in order to be completely run using the system resources (mainly CPU resources). However, in current operating systems, it is necessary to keep multitasking and interactivity. This means allowing tasks to use the system resources with the maximum fairness in order to show users high interactivity and low waiting times in applications.

The problem presented in this work does not have real-time requirements. Services do not require a deadline in order to be executed. This is why current real-time schedulers are not suitable algorithms for this problem. Services establish temporal commitments with their clients, assuming the obligation of completing the service before the negotiated time. This commitment is given to the best bidder, who has probably offered the lowest service time, so it is important for the service provider to satisfy its commitments in order to keep the confidence of its customers. This is why the service provider must make a precise and correct prediction of when the service is going to finish, which implies not only knowing the service WCET but also the workload of the system in the present and throughout the execution of the service.

Making this kind of prediction is quite difficult because what the workload of the system is going to be in the future is not known, since the arrival rate of service requests is unknown in an open system such as this. Therefore, the question we must answer is how you can ensure that a service will have enough time in the processor to finish its execution when it is not known how many services will be running at the same time.

In this work, we refer to the instant of time when the task will finish in the worst-case as the **Deadline** of task τ (D_τ). This is different from the WCET, which is the amount of time that the task will be running in the CPU in the worst-case, instead of the instant of time. We also refer to the instant of time when the task is activated and therefore is ready to run as t_a.

3.2. Deadline Prediction Scheduler with Processor Booking (DPS)

In this section we present the first iteration of a scheduler with deadline prediction using processor booking, called the Deadline Prediction Scheduler (DPS).

The main basis of this algorithm comes from a gap between fair schedulers and non-preemptive schedulers. In a system where any job or service can be accepted at any time in which we also want to keep interactivity without breaking our temporal commitments, we need to ensure that the minimum resources needed to fulfill a commitment are preserved. We accomplish this by means of **processor booking**.

In order to fulfill a temporal commitment, the scheduler needs to ensure that the task will have enough time in the processor. The processor time is a valuable resource and is directly related to the moment when a task will finish its execution. Thus, if we make a booking of the processor resource that could never be diminished when we establish a temporal commitment, we can ensure that the temporal commitment will be fulfilled. This technique is known as *Guaranteed Processor* [Jones et al. (1997), Leinbaugh (1980)]. However, Guaranteed Processor techniques are usually applied to hard real-time systems and using pre-computed schedulability graphs or acceptance tests.

Note that if we book 100% of the processor for a task, we get a FCFS scheduler again with no interactivity or multi-tasking capabilities, which is not desirable. The solution for this problem is to book a percentage of the processor that is ensured during the life of the task. The percentage of processor time is dynamically delivered in order of arrival. For example, the first process is assigned 50% of the processor, the second one 25% and so on. You can choose any kind of distribution to deliver the percentage of the processor: the shown example uses a distribution of $\frac{100}{2^n}$, where n is the order of arrival that is used to assign the processor time in *slots*. Another approach could be to always share the same amount of processor (i.e., 10%), which would be more similar to fair schedulers like Round Robin or CFS. The main difference with fair schedulers is that if we had established the commitment of booking a percentage of processor for a task, this percentage will never be lower, no matter how many tasks arrive later.

The **granularity** of the processor time is limited, so the scheduler can apply any policy that it considers appropriate when there is no more place for new tasks: to delay the moment when the task switches to the *prepared* status or even to reject the admission of the task.

With this algorithm, the scheduler can provide a Deadline prediction ($\mathcal{D}$) based on equation (1), where P_i is the position of the task i based on its order of arrival and $\mathcal{WCET}_i$ is the worst-case execution time of the task i. This is obviously a very pessimistic calculation since it does not take advantage of the **gain time** that is not used when there are not enough tasks to overload the processor.

$$\mathcal{D} = 2^{P_i} * \mathcal{WCET}_i \tag{1}$$

In this work, the *gain time* is defined as the time that is obtained when a task finishes its execution before its WCET. This time can be reused to give extra time to the rest of the tasks. In this work, the slack processor time that is not booked by any task and can be reused when all the tasks have spent all their booked time is also part of the gain time.

The gain time is used to anticipate the execution time of those tasks that have consumed all their booked execution time. For example, if there is only one task running in the processor, it has reserved a slot of 50% of the processor time and its deadline is estimated as $2^1 * \mathcal{WCET}_i$. However, since the gain time is used for the only task running in the processor, its real utilization of the CPU is 100%.

The scheduler can use the policy that it considers the most appropriate to share the remaining processor time. This *gain time policy* could be a random policy, a round robin policy, an earliest deadline first policy (EDF), or whatever the designer chooses. This policy can also be changed dynamically according to the workload or the needs of the OS.

This DPS algorithm provides a deadline prediction with the full assurance of being correct, subject to compliance with the reserves based on the order of arrival and the correct calculation of the WCET. However, this value is still too pessimistic since it ignores the relationship between the running tasks and the gain time extracted from the slack. Nevertheless, this algorithm has a time complexity of $O(1)$ for the *append* and *retrieve* functions. These functions are used to insert a new task in the scheduling queue and to remove a task from the scheduling queue. This time complexity is obtained because it only needs to push or pop tasks from the slot queue without going through the entire queue. The deadline prediction is also constant due to the simplicity of the equation (1). Finally, the *sched* function, which selects the next task to be executed, is also constant. This is because the hard work has been done in the *append* function and the tasks are scheduled following the percentage of processor.

3.3. DPS with Processor Booking using Dynamic Priority Promotion (DPS-Dy)

Processor booking is a good approach for solving the Deadline Prediction problem, but it has some drawbacks that must be resolved. It is not possible to improve the deadline prediction by assuming how much gain time the task is going to be able to use because we do not know the rate of arrival of new tasks. This algorithm is based on ensuring that the booked processor is available during the execution of the task, and it is also based on having resources available for new incoming tasks whenever possible, i.e., when the CPU utilization is not 100%. In that case, the task will have to wait until the resource is released. In this case, tasks should wait until a priority slot is released.

We define a **priority slot** as a CPU time percentage that can be booked. This value indicates the amount of CPU time that a task will be able to run in a temporal window. Thus, a 50% value will have a higher priority than a 25% one.

However, there is one improvement that can be taken into account in order to make the deadline calculation less pessimistic. This improvement is to calculate not only when a task with a higher priority slot starts, but also to anticipate when this task is going to end. In this way, when a task ends and there is not another task scheduled to grab that slot at the same moment, you can try to promote a task that is in a lower priority slot to the processor slot that has just been released.

With this technique, the scheduler assigns dynamic priority slots to every task at each moment (depending on which slots are available) and just making sure that there is never a high priority slot available when there are tasks in lower priority slots running. This promotion of priorities helps the scheduler to make more realistic and optimistic deadline predictions than those made with just the equation (1).

This algorithm makes the CPU booking by representing the use of the processor in a timeline. This timeline is divided into temporal windows for performance reasons and stores a queue with the booked tasks at that moment at each step. These tasks are ordered by priority slots, where the position in the queue is what indicates their slot. As shown in Algorithm 1, appending a new task for scheduling is to go over the timeline from the activation time t_a until the WCET has run out. This algorithm does this job by appending the new task to each priority queue of each temporal window and decreasing the task WCET according to the priority of the assigned slot and the window size. For example, given a task τ with a $WCET = 20$ and a window size $\omega = 10$, if we append τ in a window at position 1 (i.e., with the highest priority) its booked processor time will be 50%. This means that τ will be able to use half of that window time, so the algorithm decreases its WCET by $\frac{\omega}{2}$. This process is repeated along the timeline until the WCET has run out. The index of the window when the WCET is 0 is the deadline of the task. To calculate the instant of time when the task finishes, we need to multiply the window index by the window size since the window size is constant.

To better understand the proposed algorithm, we present an example

```
Input : A task $\tau$ with a worst-case execution time $WCET_\tau$
Input : Initial temporal window $t_a$ to run task $\tau$
Input : Window Size $\omega$
Output: The Deadline $D_\tau$ for task $\tau$
0.1  Window $\leftarrow t_a$;
0.2  if Window $\leq$ CurrentWindow then
0.3      Window $\leftarrow$ CurrentWindow $+1$
0.4  end
0.5  while $WCET_\tau > 0$ do
0.6      $D_\tau \leftarrow$ Window;
0.7      P $\leftarrow$ EnqueueTaskAtWindow($\tau$, Window);
0.8      $WCET_\tau \leftarrow WCET_\tau - \frac{\omega}{2^P}$;
0.9      Window $\leftarrow$ Window $+1$;
0.10 end
```
Algorithm 1: Deadline Prediction Scheduler: Append Function

trace where we append a new task to be scheduled in the system. The append function is used when a new task arrives to the system and must be prepared to be run (by being placed in a priority queue or whatever the scheduling algorithm needs to do). In our case, the append function is particularly important because it is the moment when the deadline prediction is calculated. It is also important since this is the most complex function of the scheduler in terms of time.

Suppose there are three tasks already scheduled in the OS called τ_1, τ_2, and τ_3. They are scheduled in the timeline as shown in Figure 1 with a window size $\omega = 8$. The initial scenario is defined as follows:

Task	t_a	$\mathcal{WCET}$	$\mathcal{D}$
τ_1	1	16	4
τ_2	2	14	6
τ_3	6	10	8

In this scenario with three scheduled tasks, we append a new task to book the necessary resources and to calculate its deadline. The new task (τ_4) has a $WCET_{\tau_4} = 16$ and its t_a is 4. The append function trace for task τ_4 is what follows (Figure 2):

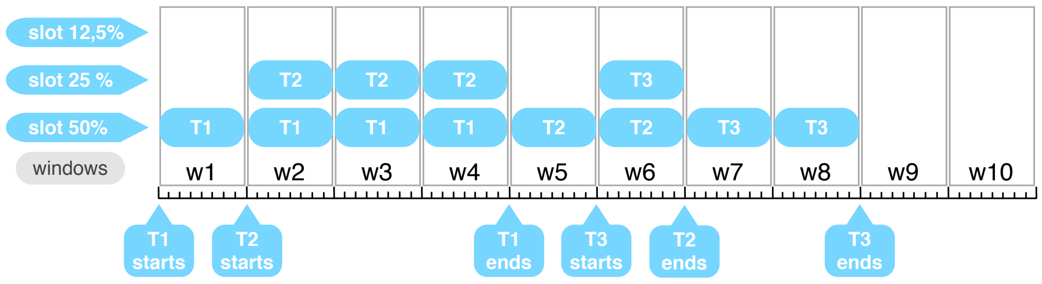


Figure 1: DPS-Dy Append Task Example: Initial Situation

1. First τ_4 is enqueued at w_4. Since there are two tasks at that window, τ_4 is appended in a priority slot of 3. With this slot and a window size $\omega = 8$, the WCET is decremented to $WCET_{\tau_4} = WCET_{\tau_4} - \frac{\omega}{2^P} = 16 - \frac{8}{2^3} = 15$.
2. At w_5, the task τ_4 is promoted to a slot of priority 2, so the remaining WCET is now $15 - \frac{8}{2^2} = 13$.
3. At w_6, the task τ_4 is enqueued again in a slot with priority 3, so the WCET is now $13 - \frac{8}{2^3} = 12$.
4. At w_7 and w_8, the task τ_4 is enqueued with priority slot of 2. After these two steps, the remaining WCET is $12 - \frac{8}{2^2} - \frac{8}{2^2} = 8$.
5. At w_9, the task has obtained the highest priority slot. Its WCET is now $8 - \frac{8}{2^1} = 4$.
6. Finally, at w_{10}, the task again has a priority slot of 1, the WCET is now $4 - \frac{8}{2^1} = 0$ and, since the remaining WCET has just reached 0, the append function is finished and the deadline is $D_{\tau_4} = 10$.

The improvement gained by the DPS scheduler using Dynamic Priority Promotion is remarkable with respect to the algorithm without priority promotion where the deadline was calculated with equation (1). Note that with the related equation the deadline would have been calculated taking into account only the slot acquired at the moment t_a (which was priority 3). Therefore, the deadline would have been $t_a + 2^P * WCET_{\tau_4} = 4 + 2^3 * 16 = 132$, which is clearly much more pessimistic than the value calculated with dynamic priority promotion.

With regard to the computational complexity: the append function has a linear cost since it has no nested loops or complex operations at each

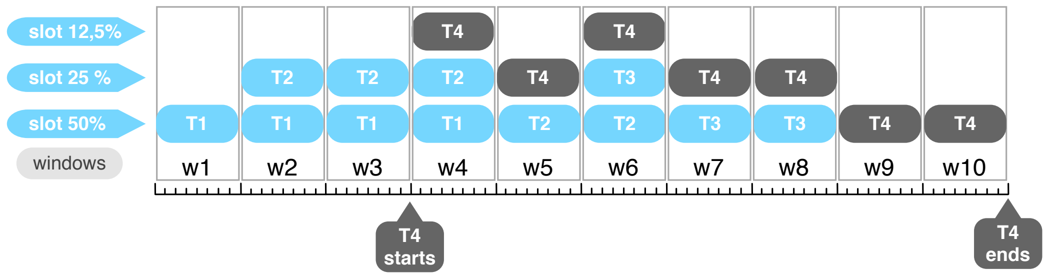


Figure 2: DPS-Dy Append Task trace example

iteration. Thus, the append function for the Deadline Prediction Scheduler is very fast.

Finally, the *schedule* function is run whenever the scheduler is invoked to select and dispatch the new task that must be introduced in the processor. This function aims to select one of the tasks that are booked for the current active temporal window. The scheduler needs to go over the queue of the active window and select a task that still has booked time to be consumed. If all the tasks have consumed their booked time, the scheduler is in gain time and can select a task using the method that it prefers, i.e. selecting a task randomly or using techniques like SJF (Shortest Job First), EDF (Earliest Deadline First), or RR (Round Robin). The computational complexity of the Schedule function is $O(n)$, where n is the number of tasks scheduled in the current active window.

With this algorithm, we have presented a more accurate deadline prediction, although we still can not predict how the gain time is going to be used. This is because we do not know the arrival rate of the tasks. However, we can still make another refinement of the algorithm to assign the percentage of processor in a more intelligent way. The next iteration of the scheduler will use a more informed assignment of the resources using the information provided by the tasks.

3.4. DPS with Benefits-based Reasoning (BDPS)

Sharing the processor resource using the order of arrival as a criterion is not always very fair or necessarily rational. Since there exists useful information that can be retrieved from the client agents that are asking for a task execution, the scheduler can use that information to make a better schedule plan. We define a better schedule plan as a distribution of the resources

(CPU time slots) that maximizes the benefit that the operating system can get from the execution of its clients tasks for the service providers. This benefit is measured in quantitative terms and is the price that clients must pay for the execution of the services that they are invoking, which is parametrized depending on the time it took to run the services.

The order of arrival is not necessarily a good criterion for distributing the CPU resource because there are some situations where some privileges should prevail. An example of those privileges is the importance of the client, which we call *priority* (Pr). When a client asks for an execution, it is not the same if that client is the system administrator or a remote client that you don't trust very much. It is not just the hierarchy of the client that is relevant but also the importance or urgency of the task. Priority-based scheduling is widespread in preemptive schedulers that expel tasks that are running if a higher priority task enters the system. However, in the scheduler that is presented in this work, the priority is used to decide how much processor time the client is able to book. For example, the scheduler may decide that a low priority task can book a slot of, at most, 25% of processor time. Critical tasks such as those that come from the operating system itself or from the system administrator may book the biggest processor slot. Even so, tasks will consume the gain time left over after all the booked slots have run out.

This reasoning method can still be improved further by introducing a benefit model in the system. This scheduler is designed for a service-based OS where several agents provide services and other agents consume those services. It is very reasonable to suppose that a service provider will not offer their services for free. The logical thing is to ask for some compensation, be it financial or other. In this work, we refer to these compensations as **benefits**.

In this scenario, the scheduler may try to maximize its benefits by prioritizing those tasks that offer a higher gratification. It is even possible for the benefit to degrade over time and the scheduler may use a reasoning process to decide not only which process to prioritize but also at what point the reported benefits are no longer interesting.

The benefit that a service execution will provide to the OS can be modeled with a decreasing function; for example, a linear decreasing function $f(x) = b - ax$ or an exponential function $f(x) = b * e^{-ax}$, where x is a variable that indicates the instant of time and a and b are constant parameters provided by the client that represent the gradient and the point of intersection between the graph of the function and the y-axis. In order to have a benefit that is greater than 0, the b constant must be also positive. The client is able to specify any other function to calculate the benefit (e.g. a constant, a step function, a WCET-based, etc).

Thus, the scheduler calculates the percentage of processor time (Ψ) that a task deserves using equation (2), where Pr is the assigned priority (a lower Pr value indicates a higher priority), Pr_{max} is the greatest value of priority, and b_max is the greatest value of benefit established by the OS itself.

$$\Psi = \frac{Pr_{max} - Pr}{Pr_{max}} * \frac{b}{b_{max}} * (1 - a) \tag{2}$$

The Ψ value does not take into account what percentage of free processor time is available at the current temporal window (φ), so Ψ is adjusted as described in equation (3). Finally, the append function for the Deadline Prediction Scheduler with Benefits-based Reasoning is shown in Algorithm 2.

$$\Psi = min(\varphi, \frac{Pr_{max} - Pr}{Pr_{max}} * \frac{b}{b_{max}} * (1 - a)) \tag{3}$$

Next, we present a trace example to show how several tasks are appended to this scheduler and how their processor percentage and deadline are calculated. We use equation (3) to calculate the percentage of CPU assigned to each task and an exponential function to calculate benefits ($B(x) = b * e^{-ax}$). In this example, we assume that $Pr_{max} = 20$ and $b_{max} = 1000$. The initial scenario is defined as follows:

Task	t_a	$\mathcal{WCET}$	Pr	a	b
τ_1	1	16	8	0.001	900
τ_2	2	14	1	0.0	300
τ_3	4	10	3	0.005	800

1. First, τ_1 enters the system. Applying equation (3), we calculate $\Psi = \frac{20-8}{20} * \frac{900}{1000} * (1 - 0.001) = 0.53946$. Since the window size is $\omega = 8$, the assigned slot for this task is $\lfloor \omega * \Psi \rfloor = 4$.
2. Thus, task τ_1 has 4 quantums guaranteed in each window, so its deadline is the end of the fourth window $D_{\tau_1} = 5$. If the task finishes at that moment ($x = (D_{\tau_1} - 1) * \omega = 32$), its benefit will be $B(32) = 900e^{-0.001*32} = 871.65$.
3. At ω_2, task τ_2 gets activated. Ψ is $min(\varphi, \frac{20-1}{20} * \frac{300}{1000} * (1 - 0.0)) = min(1 - 0.53946, 0.285) = 0.285$. This gives slot of 2 to task τ_2, while the new φ is $1 - 0.53946 - 0.285 = 0.17554$.

```
Input   : A task $\tau$ with a worst-case execution time $WCET_\tau$
Input   : Initial temporal window $t_a$ to run task $\tau$
Input   : Window Size $\omega$
Input   : Benefit constants $a$ and $b$
Input   : Maximum benefit $b_{max}$
Input   : Maximum priority $Pr_{max}$
Output: The Deadline $D_\tau$ for task $\tau$

1.1  Window $\leftarrow t_a$;
1.2  if Window $\leq$ CurrentWindow then
1.3      Window $\leftarrow$ CurrentWindow $+1$
1.4  end
1.5  while $WCET_\tau > 0$ do
1.6      $D_\tau \leftarrow$ Window;
1.7      Pr $\leftarrow$ getPriority($\tau$);
1.8      $\varphi \leftarrow$ getFreeSpace(Window);
1.9      $\Psi \leftarrow min(\varphi, \frac{Pr_{max}-Pr}{Pr_{max}} * \frac{b}{b_{max}} * (1-a))$ ;
1.10     if $\Psi > 0$ then
1.11         EnqueueTaskAtWindow($\tau$, $\Psi$, Window);
1.12         $WCET_\tau \leftarrow WCET_\tau - \lfloor \Psi\omega \rfloor$;
1.13     end
1.14     Window $\leftarrow$ Window $+1$;
1.15 end
```

Algorithm 2: DPS with Benefits-based Reasoning: Append Function

4. With 2 quantums guaranteed at each window and a $\mathcal{WCET} = 14$, τ_2 will need at most 7 windows to finish its work. Thus, its deadline is $D_{\tau_2} = 9$ as shown in Figure 3.
5. Finally, τ_3 is activated at ω_4. The percentage of processor assigned is $\Psi = \frac{20-3}{20} * \frac{800}{1000} * (1-0.005) = 0.6766$. However, since the value $\varphi > \Psi$ at ω_4, the percentage of processor is $\Psi = \varphi = 0.17554$ at the first window. At ω_5, the value $\varphi = 1 - 0.285 = 0.715$, so $\Psi = min(0.715, 0.6766) = 0.6766$.
6. The window ω_4 is now fully booked, so the quantums assigned to τ_3 are all the remaining quantums, which are 2. From ω_5 until the end of its $\mathcal{WCET}$, the guaranteed processor is 0.6766, which is translated to

5 quantums. This means that the deadline of τ_3 is $D_{\tau_3} = 7$.

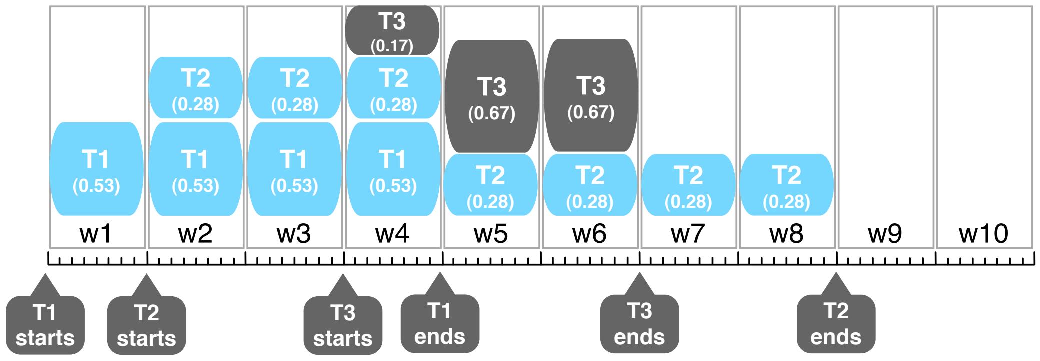


Figure 3: BDPS Append Task trace example

As this section shows, the BDPS scheduler is an algorithm that tries to maximize the benefit to be drawn from the execution of tasks, while at the sometime keeping the constraints that are desirable (i.e. the possibility of sharing the CPU fairly among all tasks and calculating an accurate deadline prediction to establish a temporal commitment with the client).

4. Experiments and Results

In order to evaluate the new scheduler presented in this work, a discrete simulator has been developed to test all the features and advantages provided by an Operating System that implements the benefits-based scheduler with deadline prediction. The operating system simulator allows us to test the functionality proposed by this work but avoids the complexity of developing the full operating system low-level abstractions. This simulator will help us to evaluate how the scheduler behaves in a distributed service-based environment where time and quality parameters (measured as benefits) are a key factor.

In this section, we present how the simulator works and the different tests that have been done by analyzing the different approaches and how they contribute to the operating system behavior. We will also compare these results with a representative set of some of the most well-known scheduling techniques.

4.1. The simulator

This operating system simulator software represents a distributed environment according to the Distributed Goal-Oriented Computing paradigm followed by the OS [Palanca et al. (2012)]. Inside this environment, there are several goal-oriented operating system nodes represented that offer services to the rest of the network. Each OS node also implements the different versions of the scheduler presented in this work. To discover and use these services, the OS uses a common publish-subscribe protocol. Some of these protocols are ZeroConf [Guttman (2001)] and XMPP PubSub [Millard et al. (1999)]. The XMPP PubSub protocol is used by this simulator to allow agents to be subscribed to a service directory. Thus, agents are notified when a service that matches their preferences is registered or unregistered.

Communications between nodes of the network are a key factor in allowing temporal commitments when invoking services. The best case for this specific-purpose operating system is a time-bounded network where communication lapses are predictable. For this reason, the communication module includes a message-passing system that simulates a time-bounded network, which allows end-to-end operations to be predicted. Providing guaranteed latency times in communications to establish message-based protocols in distributed environments was developed in works by [Tindell et al. (1995)] and [Schmidt and Kuhns (2000)].

Although clock synchronization is not a hard restriction for this simulated OS, it is a desirable option to simplify the management of temporal commitments between nodes. The simulator assumes a temporal-bounded environment which requires clock synchronization and predictable communications with respect to time. Several proposals are currently using a global time service which synchronizes the clock in every OS in the network [Kopetz and Ochsenreiter (1987)]. With these proposals, when a new operating system is added to the network, it automatically gets synchronized with the rest of the network nodes. This global time service is very useful for establishing proper temporal commitments and uses known solutions for clock synchronization in distributed real-time systems.

However, an environment with time-bounded communications is not always possible. In a real scenario where the network communication cannot be temporal-bounded (e.g. the Internet), time predictions will be less reliable. However, in the OS, prediction failures are not critical. These failures have the side effect of diminishing the trust that the client will have in the OS in the future.

This OS simulator allows us to set some parameters that change the conditions of the OS environment or the scheduler configuration. We can change the schedule policy, the number of concurrent tasks, the size of temporal windows and quantum bursts, etc.

The simulator has a scripting system that loads a configuration for the desired environment. The script can define the initial configuration of the environment (number of nodes, agents, distribution of the services by agent, etc) by setting up the scenario that is desired for the simulation. It can also schedule different events that will be processed during the simulation in order to change the environment at runtime. The scripting system is also able to generate a random task load for the OS. This is useful when we want to evaluate the same scenario with as many task loads as possible.

All the experiments were conducted using the same methodology and with at least 1000 repetitions. We found that the standard deviation converges after 1000 repetitions. The t-student *test of significance* was applied to the results of the experiments. This test aims to prove that the differences between the approaches applied in the experiments are statistically significant. The probability of obtaining equal results for different approaches (which is our *null hypothesis*) is very low (it was always below 0.05).

All the experiments were run in the same scenario. This scenario was configured as follows:

- Two OS nodes: one called *ClientHost*, which hosts a *ClientAgent*, and one called *ProviderHost*, which hosts a *ProviderAgent*.
- ProviderAgent is an agent that offers up to 500 different atomic services. Since services are atomic services, each one will be represented by the scheduler as a task.
- ClientAgent is an agent that invokes each one of the registered services. The system load is activated at the beginning of the experiment. This means that the complete set of services is activated at the beginning of the experiment.

Some simulator parameters were changed during the experiments, depending on the nature of the experiment. However, they had default values which were used when not otherwise specified:

- The WCET of each service was assigned following a Gaussian distribution between 1 and 200 time units.

- The default temporal window size was set to 256.
- The default size of the quantum burst was set to 64.
- The benefit function was an exponential function ($f(x) = b * e^{-ax}$), where the a and b parameters were assigned following a Gaussian distribution. It was mandatory for the a parameter to be in the range $[0 - 1]$. The b parameter was in the range $[1 - 500000]$, which was high enough for the purposes of these experiments. The exponential function is used for purposes of clarity and simplicity. The selected benefit function defines the curve which makes fall the benefit value. Since the a and b parameters also affect the benefit fall, we have used the same function for all nodes in order to keep things simple in the analysis of the experiment. In this way, any other function could be used in the experiment. This would change the global values of benefit but it would not alter the conclusions obtained in the experiments, because the function does not modify the behavior of the scheduler.

The set of tests performed in this work are presented below. They were divided into four test suites. The first experiment analyzed the three refinements of the scheduler to test how accurate their deadline prediction was. The second experiment focused on the benefits feature, comparing the three schedulers with some well-known schedulers. In the third test suite, we present an experiment that analyzed some relevant scheduler metrics like utilization, throughput, and measurements of time, comparing them with other schedulers. Finally, the last experiment focused on the BDPS algorithm in an attempt to find the parameters that improve the behavior of the scheduler.

4.2. Experiment 1: Deadline Prediction Accuracy

This first experiment was designed to compare the three refinements of the Deadline Prediction Scheduler. The main difference between them is how accurately they predict the instant of time when a task is going to finish, which we call its *deadline.* Each one of the refinements incrementally includes all the features of its predecessor. This means that DPS-Dy includes Resource Booking and adds the Dynamic Priority Promotion feature, while BDPS includes both Resource Booking and Dynamic Priority Promotion and adds the Benefits-Based reasoning.

The three approaches of the scheduler (DPS, DPS-Dy and BDPS) were compared in this experiment in order to prove that the accuracy of the deadline prediction improves with Dynamic Priority Promotion.

This experiment ran the simulation with different numbers of services (from 10 to 500 in steps of 10) and compared the difference between the real execution time and the deadline that was predicted. The results are shown in Figure 4. This graph shows how low the accuracy of the DPS algorithm is since it situates the deadline in the worst case, without taking into account the possible interactions between tasks. However, the DPS-Dy and BDPS algorithms made a better deadline prediction. This is because they precisely calculated when each task was going to be finished by taking into account the Dynamic Priority Promotion mechanism. DPS-Dy made the promotion by looking at the order of arrival of the tasks, while BDPS used the guaranteed percentage of CPU based on the supposed benefits. The slight differences between the prediction and the real execution time are due to the gain time bursts. When all the booked CPU was consumed and the temporal window had not yet finished, the scheduler entered in *gain time* mode, where the execution of active tasks was advanced using a different policy. In the experiments of this work, we used a gain time policy that we have called *Highest Benefit First* (HBF). This policy selects the task with the highest benefit in order to obtain the CPU during the next quantum burst.

The worst case where the DPS algorithm calculates the deadline is when all the CPU time is full and it never enters in gain time. This means that all tasks get activated at the same time and they also have the same WCET. But this worst case is a very rare situation. What is usual is that a task that finishes its execution leaves the booked CPU it had and then other tasks are able to use it. The DPS-Dy and BDPS algorithms can make a more accurate deadline prediction since they use the Dynamic Priority Promotion technique presented in this work.

The results of this experiment show us that the approach of the DPS algorithm is a very pessimistic approach since it gives us a very high and unreal deadline prediction, especially when the number of tasks grows. On the other hand, DPS-Dy and BDPS have a very similar and acceptable prediction since it is very close to the real execution time. This experiment proves that the Dynamic Priority Promotion approach improves the deadline calculation accuracy.

We have to take into account that we do not know the arrival rate of the tasks, which makes it difficult to create an off-line schedule plan without

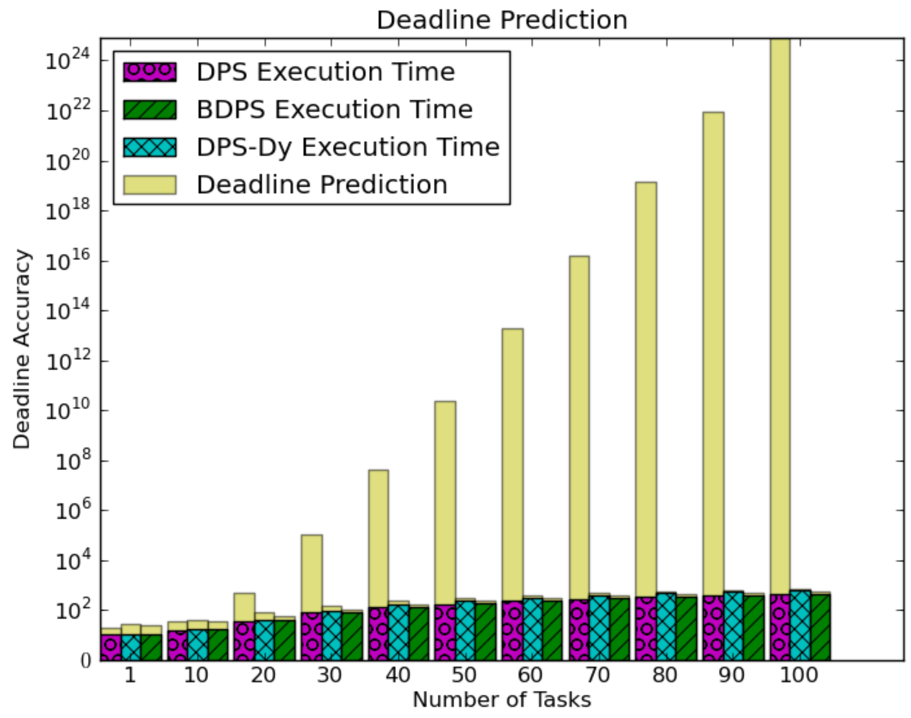


Figure 4: Deadline Precision Accuracy

breaking our commitments; however, the CPU booking technique solves this problem on-line. Moreover, the advantage of this scheduling algorithm is that, even if we have lots of gain time due to a low task arrival rate, the scheduler will still be able to reach a better execution time. In any case, the real execution time of the task will always be below the deadline prediction.

4.3. Experiment 2: Benefits

This experiment was designed to test how a benefit-oriented algorithm helps the server to reach the highest rates of benefits. It aims to compare the behavior of the three schedulers presented in this work (DPS, DPS-Dy, and BDPS) with a representative set of well-known operating system schedulers.

We briefly present these scheduling algorithms:

FCFS *First-Come First Served* algorithm [Silberschatz et al. (1998)] is a non-preemptive scheduler that gives the CPU to the tasks in order of arrival.

When a task gets the CPU, it keeps the processor busy until it finishes its execution.

RR *Round Robin* [Silberschatz et al. (1998)] is an algorithm made for sharing the CPU with the same percentage of time. Round Robin splits the time in quantums and gives the CPU cyclically to each task during the duration of a quantum burst.

CFS The *Completely Fair Scheduler* [Wong et al. (2008)] is an algorithm that tries to make the most fair use of the CPU. It works by giving the CPU to the task that has used the least amount of time. This way it maximizes the interactive performance, while keeping a high CPU utilization.

SRTF *Shortest Remaining Time First* [Silberschatz et al. (1998)] is an evolution of the well-known SJF algorithm (Shortest Job First). This scheduler selects the task with the smallest execution time to execute next. SJF is a non-preemptive scheduler, while SRTF is the preemptive variant that takes into account the remaining execution time instead of the full execution time. SRTF makes the decision of assigning the CPU when a task arrives or completes. SJF and SRTF achieve better performance than FCFS. However, unlike FCFS, there is the potential for starvation in these schedulers. Starvation occurs when a large process never gets run because shorter tasks keep entering the queue.

We also compare the schedulers with an algorithm developed with the single purpose of optimizing the benefit. We have called this algorithm the *Highest Benefit First* (HBF) scheduler. HBF is an algorithm that selects the task with the highest benefit for execution. This is a non-preemptive scheduler which was originally designed for the gain time stage of the BDPS algorithm instead of using approaches unrelated to the benefit like FCFS, random. etc. However, for experimental purposes, we developed a full scheduler using HBF in order to compare it with the rest of the schedulers.

The experiment under consideration consisted of a set of tasks executed by all these algorithms. The system workload was the same for every algorithm, so the only change was the order in which tasks were executed (depending on each algorithm's policy). At the end of the experiment, it calculated how much benefit the provider agent acquired at the server OS.

In this experiment, each task used an exponential benefit function ($f(x) = b * e^{-ax}$), where the a and b parameters were assigned following a Gaussian distribution. The a parameter was always in the range $[0 - 1]$. For the b parameter, we selected a smaller range ($[1 - 500]$) in order to make the results more manageable. Every time a task leaves the CPU, it gets a benefit following the $f(x)$ function.

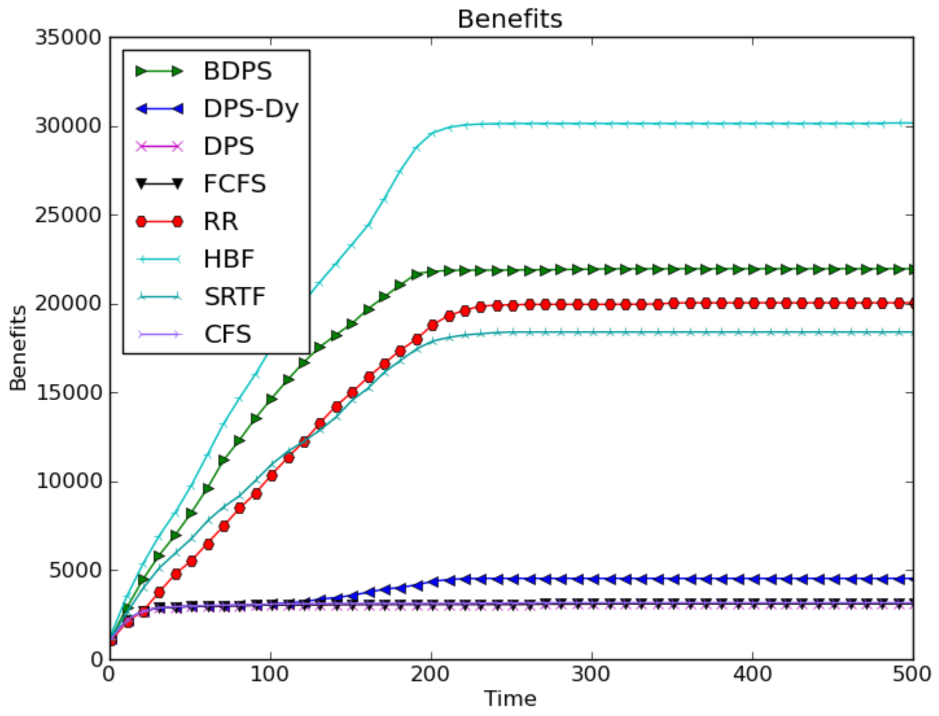


Figure 5: Benefit experiment results for each tested algorithm

Figure 5 shows the results of the experiment. This graph shows that the algorithm that reached the highest benefit rate was the HBF scheduler. This makes sense since this algorithm was optimized to always run the task that gave the most benefits in the short-term. Since each task provides a part of the benefit each time it reaches the CPU, the Round Robin scheduler (which shares the CPU in equal amounts of time between each task) also reaches a high quota of benefit.

The Benefits-based Deadline Prediction Scheduler (BDPS) meets all the desired conditions presented in this work. BDPS shares the CPU time using the benefit function of the tasks and uses temporal windows where the guaranteed processor is assigned in quantum bursts. Thus, it reaches high rates of benefits without starvation or interactive performance problems and it is also able to calculate an accurate deadline prediction. This allows the agents to establish proper temporal commitments with their clients. The SRTF scheduler has one of the highest rates of executed tasks per unit time. This makes this algorithm also reach a good benefit value, which is close to the RR rates.

The FCFS, DPS and DPS-Dy algorithms are similar in that they do not care about benefits when scheduling nor do they share the CPU time between tasks fairly. These scheduling algorithms share the CPU using the order of arrival as priority, which results in very low benefit values.

In this experiment, the BDPS reached very good values of benefit. However, there was another scheduler (the HBF one) that reached even better results when we only measured benefits. However, there were other metrics that must be taken into account, like response time, utilization, or the deadline prediction. In later experiments, this metric (the deadline prediction) invalidated the HBF option, because it must necessarily accept every task that enters the system and, therefore, share the CPU time with that new task.

4.4. Experiment 3: Scheduler Metrics

In this experiment we present some common scheduler metrics [Tanenbaum (2008)] that allow us to compare the algorithms presented in this work with some of the reference algorithms in terms of performance, utilization, etc. We selected five metrics to compare the same algorithms from experiment 2.

The metrics are the following:

Utilization This metric measures how busy the CPU is. The definition that we used in this work for utilization is presented in (equation 4), where nQ is the total amount of used quantums, nCS is the number of context switches, and cCS is the cost of a context switch. A context switch occurs every time the OS suspends the execution of a task to make way for the execution of another task. Suspending a task involves storing its state (registers, program counter, etc), which is computationally intensive. That is the reason why the OS designs aim to optimize the use and cost of context switches.

$$U = \frac{nQ}{nQ + nCS * cCS} \tag{4}$$

Throughput This metric measures the number of tasks (nT) that complete their execution per time unit (equation 5).

$$T = \frac{nT}{nQ + nCS * cCS} \tag{5}$$

Turnaround Time This is the amount of time needed to execute a particular task. This measures the time since a task was activated until it finishes its execution. This value is usually calculated as the average of all the turnaround times of every task.

Waiting Time This metric measures the time that a task has been waiting in the ready queue. It is always a lower value than the turnaround time since it includes the waiting time and the execution time.

Response Time This is the amount of time that a task takes since it is activated until the first execution is produced. In this work, we measure this value as the instant of time when a task first gets the CPU.

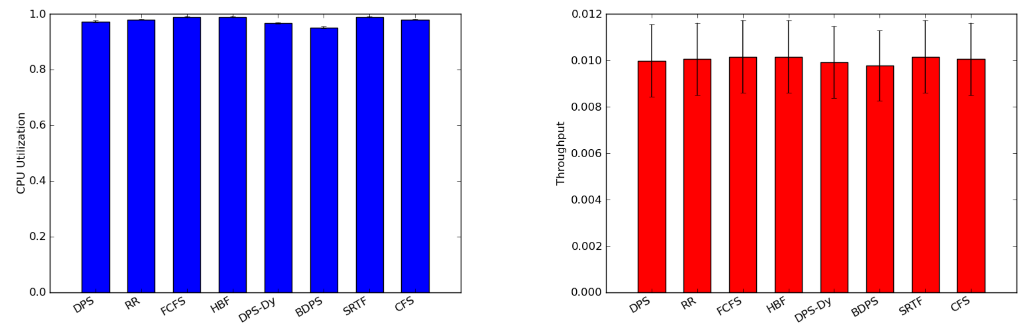


Figure 6: Utilization and Throughput

Figure 6 presents the results of the utilization and throughput metrics for each one of the algorithms. It can be observed how the vast majority of the algorithms keep a high rate of utilization since they are completely dependent on the number of context switches that are performed. The BDPS algorithm is the algorithm that performed the most context switches. This occurred because the size of the temporal window that we adopted was four times the size of a quantum burst, which produced many potential context switches inside each window because BDPS does not always use the full quantum burst size to change the running task. The booked CPU for a task inside a window may be smaller than a quantum burst (depending on the benefit it provides and its priority), so, on average, there are more context switches than using other fair algorithms like RR or CFS. However, the CPU Utilization of the BDPS algorithm is still over the 90%, which is a very acceptable rate. The algorithms with the higher CPU Utilization rate are the non-preemptive ones because they minimize the number of context switches. The Throughput metric has a very close relationship with the Utilization metric. Thus, the results are very similar to the CPU Utilization ones.

Figure 7 shows the Average Turnaround Time and the Average Waiting Time. We can confirm that the SRTF scheduler was the one that provided the best rates of turnaround time (and, therefore, waiting times) because it selects every task that is closer to its end. However, all the *fair* algorithms (RR, CFS, DPS, DPS-Dy and BDPS) that share the CPU more equitably have longer turnaround times and longer stays in the ready queue. The other non-preemptive scheduler (FCFS) and the HBF scheduler have intermediate values for the turnaround time metric.

Figure 8 shows the Average Response Time for each of the studied algorithms. The Response Time is an interesting metric that shows how fast the user may receive an interaction from the service. That is why we measure how much time a task takes to first get the CPU since the task has been activated. The group of the *fair* algorithms traditionally have a good Response Time rate because they give the same opportunities to each task. The opposite occurs with algorithms with a high starvation risk (like FCFS or HBF) since a task may be blocked for long time periods if higher priority tasks are persistently entering the system.

The BDPS has a special position in this metric since it gets a very good Response Time rate. This is because it uses temporal windows and, inside each temporal window, it splits the time into bursts with a size that depends on the benefit parameters. This is why every task has a chance, even a small

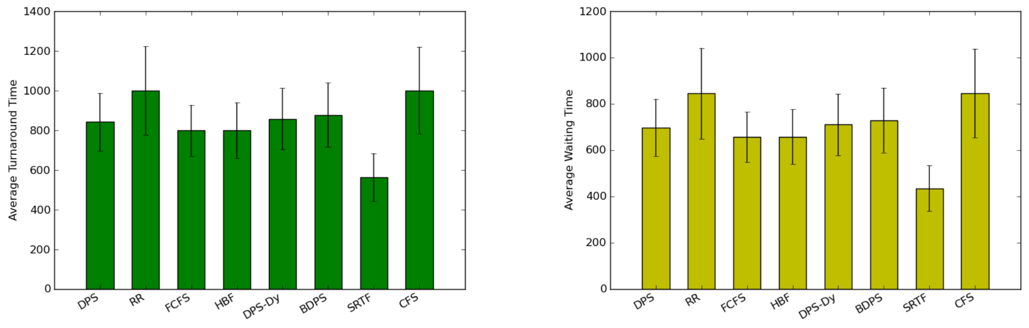


Figure 7: Average Turnaround and Waiting Time

one, to get the CPU at an early stage. Note that the DPS and DPS-Dy do not have Response Time rates that are as good as the BDPS one. This is because the DPS and DPS-Dy schedulers book the CPU always following the same method (the half of the remaining window time). This has a worse granularity and supports fewer tasks per window, which delays the entry of the last task to arrive.

Finally, Table 1 presents a qualitative comparison of all the algorithms using a normal distribution of their error differences. The table uses the following legend: a ● indicates that the algorithm has a good assessment in the metric; a ○ indicates that the algorithm has a bad assessment in the metric; a ◑ indicates that the algorithm is in the middle of the metric, which means that this scheduler is not distinguished by the metric; a ⊠ indicates that the metric is not applicable for this algorithm, which invalidates the scheduler for the purposes of the case of study of this work.

The summary table shows that there is no algorithm that gets the maximum score in all the metrics. The SRTF scheduler obtains one of the best ratings inasmuch as it presents very good utilization and turnaround values, while it still presents acceptable values for benefit and response time. However, the impossibility of calculating a deadline prediction does also invalidate this scheduling algorithm for the purposes of our work. The non-preemptive approach (SJF) would simplify the action of calculating this prediction, but nowadays a non-preemptive scheduler has the starvation and interactive performance counterparts [Tanenbaum (2008)]. In addition to the algorithms

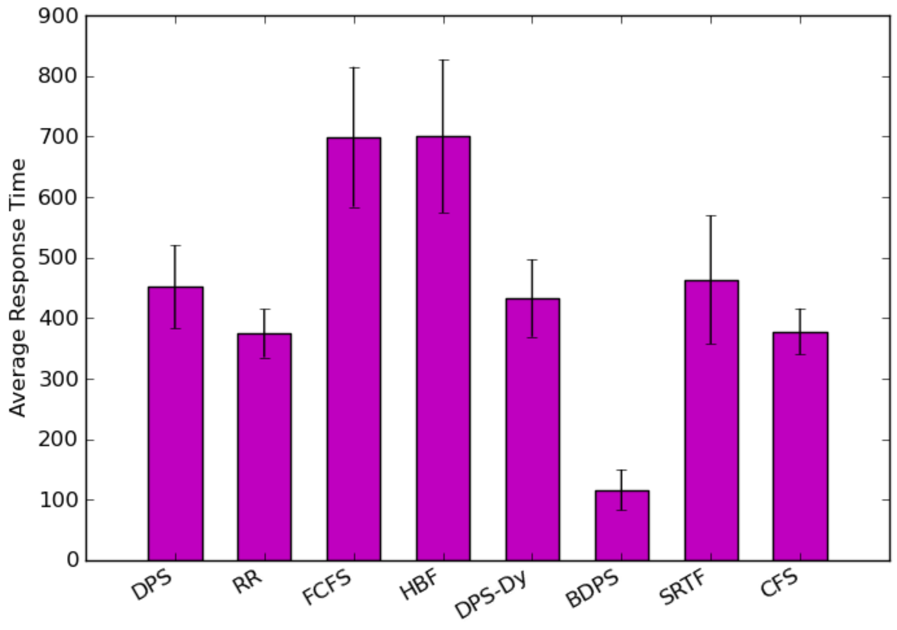


Figure 8: Average Response Time

presented in this work, the only algorithm that is capable of performing deadline predictions is the FCFS scheduler. Since it is a non-preemptive scheduler, as long as the WCET value is well calculated, it would be easy to know when a task is going to finish its execution. Nevertheless, the FCFS algorithm presents very bad results for the benefit and response time metrics.

The results presented in the experiments show us that the BDPS algorithm has good results for almost all the metrics, providing acceptable benefit results, turnaround times, and really good response times, and very accurate deadline predictions. This makes the BDPS a good choice for the operating system architecture presented in this work [Palanca et al. (2011, 2012)].

4.5. Experiment 4: Tuning the BDPS scheduler

Since the BDPS has been selected as the most appropriate scheduling algorithm for the OS architecture based on the Distributed Goal-Oriented

	Benefit	Utilization	Turnaround	ResponseTime	Deadline Pred.
DPS	○	◑	◑	◑	○
DPS-Dy	○	○	◑	◑	●
BDPS	◑	○	◑	●	●
HBF	●	●	◑	○	⊠
FCFS	○	●	◑	○	●
RR	◑	◑	○	◑	⊠
CFS	○	◑	○	◑	⊠
SRTF	◑	●	●	◑	⊠

Table 1: Comparison Table

Computing paradigm, we now focus on this scheduler. The purpose of this experiment was to tune the algorithm by paying attention to the different parameters that could modify the behavior of the scheduler. We selected two parameters that could be modified in order to change how the scheduler behaved thereby changing the results of the metrics that tell us how good the algorithm was. The two parameters that changed were the size of the temporal windows and the size of the quantum bursts.

We designed an experiment where we ran 500 tasks using different sizes for the temporal windows (from 64 to 2048). The maximum size of the quantum burst was set again to a quarter of the window size. The scenario for each test was replicated exactly, and we only changed the size of the temporal window. Figure 9 shows how the utilization and throughput metrics increased when we increased the size of the window. This is because higher values of window sizes generate fewer context switches, which leaves more time to run tasks.

Figure 10 presents the results for the Average Turnaround Times and Average Waiting Times of the same experiments. These graphs show how the average times had their lowest value for intermediate values of temporal window sizes.

It can be observed that the bigger the window is, the better the turnaround times it gets. This is because a bigger window allows the scheduler to book larger CPU time slots, which makes the algorithm like a non-preemptive scheduler. This behavior is compensated by using quantum bursts like fair algorithms (RR, CFS), which ensures that the scheduler shares the CPU time fairly and avoids starvation. However, increasing the size of the quantum burst produces the same situation as with big temporal windows.

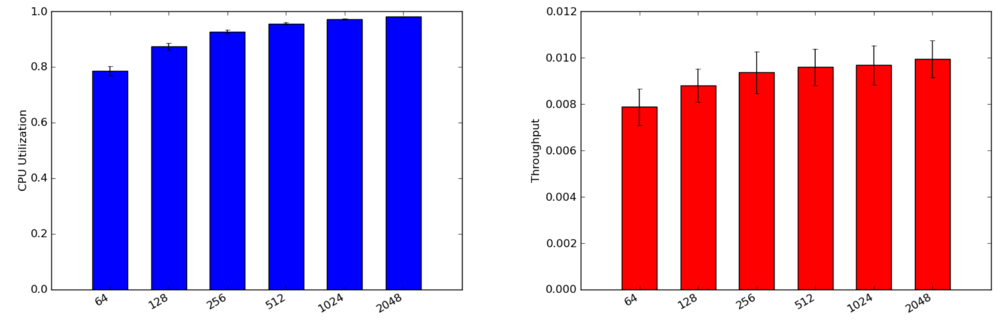


Figure 9: Window Size: Utilization and Throughput

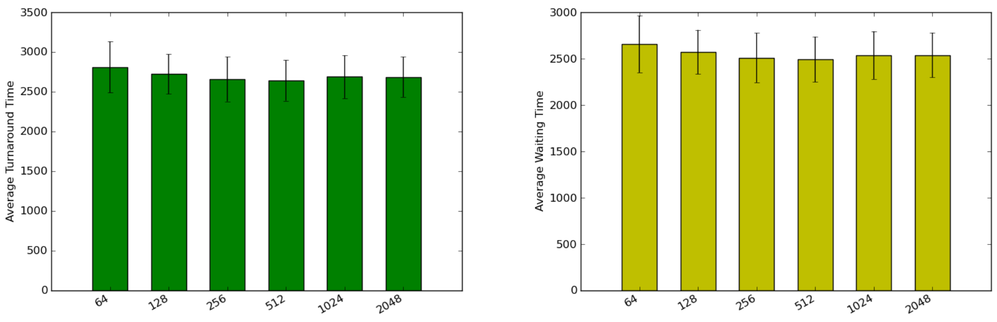
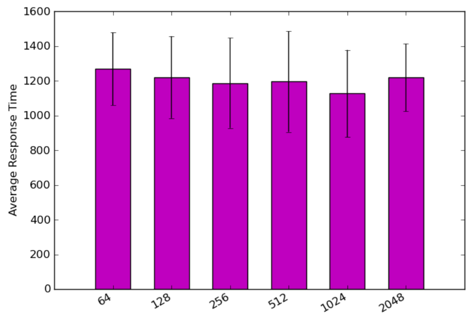
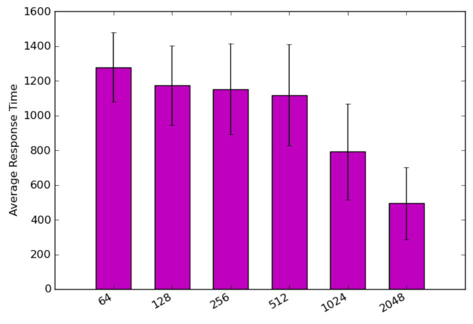


Figure 10: Window Size: Turnaround and Waiting Times

It is important to note that these values are directly proportional to the size of the quantum bursts. This means that with big temporal windows and small quantum bursts, there is a high number of context switches, what will decrease our utilization metric. On the other hand, lower response times are obtained. To compare this, we performed an experiment where we kept the size of the quantum burst constant in a small size. Figure 11 compares the Response Time for different windows with the quantum burst proportional to the window size (the value was a quarter of the window size value (a) with a constant quantum burst value, which was set to 32 (b)).



(a)



(b)

Figure 11: Window Size: Response Times varying the quantum burst size

As expected, these results show that higher values of window size and lower values of quantum burst size provided the best response times for the BDPS algorithm. However, there are some drawbacks to take into account if these values are used. The smaller the quantum burst size is, the smaller the algorithm utilization will be. And, what is most important, since the deadline prediction is always adjusted to the end of a window. This is because we predict that a task is going to finish before the end of a window, but we do not know the exact instant of time inside that window, higher values of temporal window sizes will have a worse deadline prediction accuracy (Figure 12).

In summary, in these test suites, we have experimented with how the size of the temporal window and the size of the quantum bursts directly affect the behavior of the BDPS algorithm. A good proportion between the two parameters might decrease the response time significantly. However, depending on which metric you are interested in boosting, the OS can select different values for these parameters to tune the scheduler. High values of window size provide high utilizations and throughput, but they deteriorate the deadline prediction and the turnaround and waiting times. This experiment showed that with a ratio of 1 to 4 between the window size and the quantum burst, intermediate values of window size (around 512) return acceptable average results for these metrics. This is the tuning that we have selected for the scheduler. However, the OS is able to change these parameters depending on

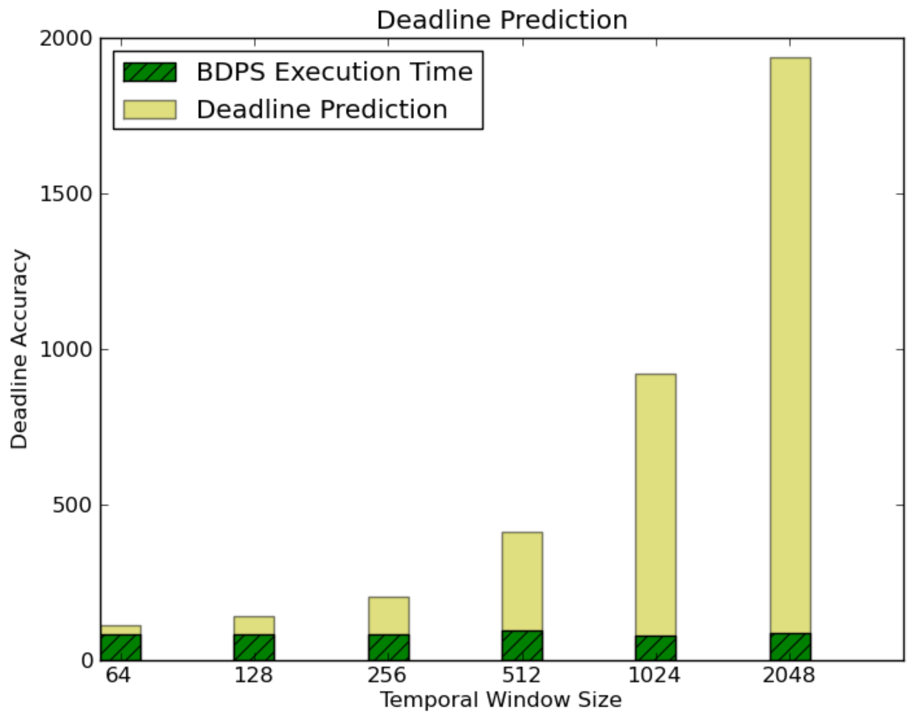


Figure 12: Deadline Prediction Accuracy for different window sizes

the environment (average WCET values, average arriving rate, etc) or the metrics it wants to boost.

5. Conclusions

This work has presented a scheduling algorithm that can be used to predict the completion time of a task in an Operating System that implements the Distributed Goal-Oriented Computing paradigm. This scheduler is useful for prioritizing tasks with functions that provide high benefits and to establish temporal commitments with the clients that invoke those tasks.

Guaranteeing the completion time of a running task is a hard task when performing this guarantee on-line and taking into account parameters provided by the task (i.e. the benefit that is obtained by running the task). The BDPS algorithm is able to create a scheduling plan that predicts when the tasks are going to be completed. This is done by using guaranteed processor techniques. BDPS books the CPU resource according to the priority of the client and the benefit function.

In summary, the experiments have demonstrated that the BDPS scheduler properly fits into a Distributed Goal-Oriented environment. It is also capable of calculating accurate deadline predictions, which allows the scheduler to establish temporal commitments with its clients. BDPS has also demonstrated a really good response time, which is a signal of how fairly the CPU is shared. The algorithm is also parameterizable (both the window size and the quantum size), which opens up the possibility of dynamically adapting these parameters according to the workload. One of the biggest advantages of BDPS is that it creates scheduling plans using the benefit parameter, which allows the OS to sort the execution of tasks, maximizing the potential benefits to be achieved. The rest of the analyzed metrics were quite similar to the other well-known schedulers.

Acknowledgments

This work is supported by the TIN2009-13839-C03-01 project of the Spanish government, PROMETEO/2008/051 project, FEDER funds and CONSOLIDER-INGENIO 2010 under grant CSD2007-00022.

References

Buttazzo, G., Stankovic, J., 1993. Red: Robust earliest deadline scheduling. In: Proc. International Workshop on Responsive Computing Systems. pp. 100–111.

Fernández-Olivares, J., Garzón, T., Castillo, L., García-Pérez, Ó., Palao, F., 2007. A Middle-Ware for the Automated Composition and Invocation of Semantic Web Services Based on Temporal HTN Planning Techniques. Current Topics in Artificial Intelligence. Springer, 70–79.

Guttman, E., 2001. Autoconfiguration for IP networking: enabling local communication. Internet Computing, IEEE 5 (3), 81–86.

Hao, J., Zhi-jian, S., 2006. The tcpn-based verification of temporal consistency in web service process. In: ICEBE '06: Proceedings of the IEEE International Conference on e-Business Engineering. IEEE Computer Society, pp. 302–306.

Herrero, A., Navarro, M., Corchado, E., Julián, V., 2011. RT-MOVICAB-IDS: Addressing real-time intrusion detection. Future Generation Computer Systems (http://dx.doi.org/10.1016/j.future.2010.12.017).

Hsu, C.-C., Huang, K.-C., Wang, F.-J., 2011. Online scheduling of workflow applications in grid environments. Future Generation Computer Systems 27 (6), 860–870.

Jones, M., Roşu, D., Roşu, M., 1997. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In: ACM SIGOPS Operating Systems Review. Vol. 31. ACM, pp. 198–211.

Kopetz, H., Ochsenreiter, W., Aug. 1987. Clock Synchronization in Distributed Real-Time Systems. IEEE Transactions on Computers C-36 (8), 933–940.

Leinbaugh, D., 1980. Guaranteed response times in a hard-real-time environment. IEEE Transactions on Software Engineering 1, 85–91.

Lv, M., Guan, N., Zhang, Y., Deng, Q., Yu, G., Zhang, J., 2009. A Survey of WCET Analysis of Real-Time Operating Systems. In: Embedded Software and Systems, 2009. ICESS '09. International Conference on. pp. 65–72.

Martin-Diaz, O., Cortes, A. R., Duran, A., Müller, C., 2005. An Approach to Temporal-Aware Procurement of Web Services. In: ICSOC. pp. 170–184.

Millard, P., Saint-Andre, P., Meijer, R., 1999. XMPP Publish-Subscribe Extension. Tech. Rep. XEP-0060, XMPP Standards Foundation.
URL `http://xmpp.org/extensions/xep-0060.html`

Naseri, M., Towhidi, A., 2007. QoS-Aware Automatic Composition of Web Services Using AI Planners. In: Second International Conference on Internet and Web Applications and Services, 2007. ICIW'07. IEEE, p. 29.

Palanca, J., Julián, V., García-Fornes, A., 2011. A Goal-Oriented Execution Module Based on Agents. In: 44th Hawaii International Conference on System Sciences (HICSS 2011). pp. 1–10.

Palanca, J., Navarro, M., Julian, V., García-Fornes, A., 2012. Distributed Goal-oriented Computing. Journal of Systems and Software (http://dx.doi.org/10.1016/j.jss.2012.01.045).

Pan, F., 2005. Temporal aggregates for web services on the semantic web. In: IEEE International Conference on Web Services, 2005. ICWS 2005. pp. 831–832.

Schmidt, D., Kuhns, F., 2000. An overview of the real-time CORBA specification. Computer 33 (6), 56–63.

Silberschatz, A., Galvin, P., Gagne, G., Silberschatz, A., 1998. Operating system concepts. Vol. 4. Addison-Wesley.

Solanki, M., Cau, A., Zedan, H., 2004. Augmenting semantic web service description with compositional specification. In: Proceedings of the 13th international conference on World Wide Web. ACM, pp. 544–552.

Tanenbaum, A. S., 2008. Modern Operating Systems. 3rd edition. Vol. 2. Prentice Hall New Jersey.

Tindell, K., Burns, A., Wellings, A. J., 1995. Calculating controller area network (CAN) message response times. Control Engineering Practice 3 (8), 1163–1169.

Wong, C. S., Tan, I., Kumari, R. D., Wey, F., 2008. Towards achieving fairness in the Linux scheduler. SIGOPS Oper. Syst. Rev. 42 (5), 34–43. URL http://doi.acm.org/10.1145/1400097.1400102