Customer Appeasement Scheduling[*]

Mohammad R Nikseresht Anil Somayaji Anil Maheshwari

Abstract

Almost all of the current process scheduling algorithms which are used in modern operating systems (OS) have their roots in the classical scheduling paradigms which were developed during the 1970's. But modern computers have different types of software loads and user demands. We think it is important to run what the user wants at the current moment. A user can be a human, sitting in front of a desktop machine, or it can be another machine sending a request to a server through a network connection. We think that OS should become intelligent to distinguish between different processes and allocate resources, including CPU, to those processes which need them most. In this work, as a first step to make the OS aware of the current state of the system, we consider process dependencies and interprocess communications. We are developing a model, which considers the need to satisfy interactive users and other possible remote users or *customers*, by making scheduling decisions based on process dependencies and interprocess communications. Our simple proof of concept implementation and experiments show the effectiveness of this approach in the real world applications. Our implementation does not require any change in the software applications nor any special kind of configuration in the system, Moreover, it does not require any additional information about CPU needs of applications nor other resource requirements. Our experiments show significant performance improvement for real world applications. For example, almost constant average response time for *Mysql* data base server and constant frame rate for *mplayer* under different simulated load values.

1 Introduction

1.1 Motivation

Almost all of the current process scheduling algorithms, which are used in modern operating systems (OS), have their roots in the classical scheduling paradigms which were developed during the 1970's. But today's computers have different types of software loads and user demands. It is not important to maximize CPU utilization, as most modern machines, either desktops or servers, have multiple cores/CPUs and most of the time they have idle CPU

[*]Technical Report TR-10-18, School of Computer Science, Carleton University. This work is partially supported by NSERC.

cycles. It is not important to minimize job turn around time, because most machines are not running CPU intensive jobs. Most machines have a varying load pattern which depends on the requests coming from local and/or remote users. Often it may not be important to maximize system throughput. Typically throughput is defined in terms of non-interactive jobs submitted to a machine, while most modern tasks have some form of interaction with users. What is important is to run what the user wants at the current moment. A user can be a human, sitting in front of a desktop machine, or it can be another machine sending a request to a server through a network connection. As we are looking at a wider range of users, we call them *customers*. In our view, local or remote customers may send different requests to a system. Due to rapid changes in customer demands and requests, the need for CPU time among different processes or groups of processes changes rapidly. Some OSs and authors have framed part of this problem as the user interactivity problem and have addressed it in several ways, but none of them have presented an easy to use solution [8, 9, 11, 13, 17]. These solutions usually need some form of configuration and/or input from the user, or need additional information from applications. We will elaborate on this in Section 1.2.

We believe a key change in OS resource management is to make them aware of what the applications are doing on top of them. In other words, we think OS should become intelligent to distinguish different processes and allocate resources, including CPU, to those processes which need them most. We think the processes that need more resources are the ones which are *externally observable* at the time of scheduling. If a customer is waiting for a response from a process, then we say that, this process is *externally observable*. If this process is waiting for a service from another process, then the second process is also *externally observable*. We believe that as a first step to make OS aware of what is happening in the system, process dependencies and interprocess communications should be considered. Unfortunately, commodity OSs do not support process dependency detection or interprocess communication detection. Although, OS kernel usually has some information about interprocess communications and process dependencies, they are generally dispersed in various unrelated kernel data structures, and the kernel does not use those information to make any process scheduling decisions or any other resource allocation decisions.

In this study we are developing a model which considers the need to satisfy interactive users and other possible remote users or customers. This model makes scheduling decisions based on process dependencies and interprocess communications. We want to develop a scheduling algorithm which tries to minimize a user's dissatisfaction or unhappiness. We call this *customer appeasement* as it is not possible to make every customer satisfied specially under heavy loads by running all processes fairly. A scheduling policy resulting from customer appeasement model is not a fair scheduling policy as it tries to find more important processes and give them more priority. This goal is achieved by a model which tracks process dependencies and communications using scalar values assigned to processes, customers and the whole system. Our simple proof of concept implementation and experiments show the effectiveness of this approach in real world applications. Our implemention does not require any specific change in the software applications or in the configuration of the system. Moreover it does not require any additional information about CPU needs of applications and other resource requirements.

1.2 Related Work

As mentioned in Section 1.1, as far as we know, there are many studies which try to solve interactive or multimedia applications scheduling problems, but none of them has a broader view of finding the optimal scheduling solution based on a well defined criteria for all applications, specially under heavy load.

Most commodity OSs use some heuristics based on process execution/sleeping behavior to detect interactive processes to increase their priority and reduce their latency. Windows [13] and FreeBSD [9] use multi-level feedback queue schedulers. In this scheme CPU-bound processes receive lower priorities and processes blocked waiting for I/O receive higher priorities. The Linux Vanilla or O(1) scheduler [11] (used in kernels before 2.6.23) has a similar mechanism. Processes with longer sleep times and shorter execution times are identified as interactive and receive higher priorities. Windows [13] adds more intelligence by differentiating processes waiting on different devices. For example, processes waiting on keyboard receive higher priority than those waiting on a disk. Etsion et al. and Yan et al. [8, 17] show that depending only on execution behavior is not adequate to distinguish interactive processes properly. Ingo Molnar, the designer of Linux CFS scheduler [10, 12] tries to mitigate this problem by not depending too much on process execution/sleeping behavior. CFS scheduler doesn't change interactive processes' priority any more, it only inserts them in front of the run queue every time an interactive process wakes up [10, 12] (also see Section 3.3).

Windows [13] also uses "windows system input focus" as a measure of user interaction and it increases the priority of a process which has the input focus. Using input focus may help to improve interactivity performance but has several problems. If a user is running multiple interactive programs, for example an audio player and a web browser, while he/she is browsing the web and input focus is on the web browser, the user still wants the audio player to play the music well. Input focus mechanism also might not be usefull if a user interacts with the system through the network.

Etsion et al. [8] use process display output production as a means of detecting interactive and multimedia applications. They schedule processes based on their display output production in a way that all processes have a chance to produce display output at the same rate. That might be usefull for multimedia applications where, for example, all video applications play at the same frame rate regardless of their window size. This approach only addresses desktop applications as any network user has no display access. Also, it might be possible that a compute intensive job creates a huge amount of disply output and receives an increase in its priority while it actually is not an interactive application.

Some researchers and OSs, allow real time or interactive processes to specify their CPU requirements and time constraints. For example in Mac OS X [14], a real time process may ask for a specific CPU requirement. Yang et al. (RedLine) [18] use almost the same principles and treat interactive processes like real time processes. In RedLine processes can ask for a specific CPU and other resource requirements. RedLine also has an admission mechanism which may not allow the process to execute as an interactive process if the system does not have enough resources as requested by the process.

Zheng et al. have an implementation called SWAP [19] which recognizes process dependencies but it does not distinguish interactive or any other type of processes which might need increased priority. It only tracks process dependencies based on system calls and prevent a high priority process being blocked by a low priority process that has locked a resource needed by the high priority process (priority inversion problem).

Zheng et al. work called RSIO [20] has the most similarities with our work. RSIO looks at process I/O patterns as a way of detecting interactive processes. It also tries to identify other processes involved in a user activity and provide a scheduling policy to improve interactive performance. This policy is based on access patterns to I/O devices. RSIO needs a configuration file that defines which I/O devices should be monitored to detect interactive processes. It has also a relatively complicated heuristic mechanism to detect processes involved in a user interaction.

1.3 Contributions

The work presented in this paper has major differences from all of the previous work.

1. We develop a model (customer appeasement model) with a criteria which tracks process dependencies and customer requests. This model gives us the ability to compare different schedulers analytically, and develop new scheduling policies based on the analytical results.

2. Our model considers all of the process communications and dependencies in the system which are indications of a customer request. Other systems e.g. RSIO [20] typically consider a subset of communications related to a subset of I/O devices.

3. The customer appeasement model objective is to improve the performance of any externally observable processes specially under heavy background loads, this includes traditional interactive processes, i.e. desktop and multimedia applications.

4. We have a simple proof of concept implementation which does not need any configuration file, or process specification information. It does not require any changes in the software applications either. It automatically and without any user assistance detects those processes which need more resources as defined in this paper, and increases their priority. This is in contrast to other work such as Redline [18], RSIO [20] or as allowed by OS X [14], which require some form of configuration or process resource specifications from user.

5. Our experimental results show significant performance improvements for both interactive applications and server processes such as *Apache* web server [1] and *Mysql* [4] data base servers. For example, we observed almost constant average Mysql response times, and almost constant frame rate for *mplayer* under different (simulated) background loads.

6. One of our goals is to make the implementation simple, easily portable to different Linux kernels and distributions, and easy to use for a novice user. In order to achieve this, we use SystemTap [6]. SystemTap is a diagnostic tool, but it has made our implementation a simple script which can be run on any SystemTap equipped distribution with a compatible kernel without the need to recompile and install a new kernel. Our script has been tested on kernel versions 2.6.31, 2.6.32, 2.6.35, but should be compatible with any kernel that has a recent version of CFS scheduler.

The rest of this paper is organized as follows: We describe the customer appeasement model and its basic definitions in Section 2. In Section 3, we compute the unhappiness values for two simple scenarios for some of the classical and modern OS schedulers. In Section 4, we propose an algorithm to use unhappiness values to change process scheduling. In Section 5, we explain our simplified request based priority elevation technique. We give a more detailed explanation on the implementation in Section 6. In Section 7, we present our experimental results, concluding remarks are presented in Section 8.

2 The Customer Appeasement Model

In this section we introduce the customer appeasement model and explain the parameters and variables in detail.

2.1 Definitions

In the customer appeasement model we use the following terms, notations and definitions:

Process: A process P is any software entity inside the system which can be scheduled to run on a CPU by the OS. (This definition includes tasks, threads or light weight processes.)

Customer: A customer C is any outside entity which can send requests to processes in the system. Customers are independent from each other. We may distinguish between local and remote customers.

Direct Request: A request R is any type of input from a customer to a process. $R^k_{i\to j}$ denotes the k^{th} request from customer C_i to process P_j.

Indirect Request: A process may receive a request from a customer indirectly. This happens when a process which has received a direct request from a customer in turn, sends a service request to another process.

Weight of a request: w^k_i is the weight or importance of the request $R^k_{i\to j}$ for the customer C_i. This may be measured or inferred from customer's behavior.

Customer weight: $W(c_i)$ is the parameter which is used to distinguish between different customers. It represents weight or importance of customer c_i for the system.

Unhappiness: $u^k_{i\to j}$ is an integer value related to the time delay that customer C_i experiences as a result of sending the request $R^k_{i\to j}$ to the process P_j.

2.2 Computation of Unhappiness Value

The amount of unhappiness assigned to a process due to a request, changes according to the rules explained in this section. The request might have been sent either directly or inderectly through another process to P_j. In the simplest situation, the unhappiness for process P_j at any moment is defined as the elapsed time since the moment process P_j receives $R^k_{i\to j}$ minus the amount of CPU time that P_j has been allocated. When P_j sends a response back to the customer C_i, then $u^k_{i\to j}$ is set to zero. Observe that:

1. The unhappiness value $u^k_{i\to j}$ increases as time passes.
2. $u^k_{i\to j}$ is decreased by the amount of time that process P_j runs on a CPU processing request $R^k_{i\to j}$.
3. When process P_j requests a service from another process P_s and blocks, then $u^k_{i\to j}$ is divided between P_j and P_s as follows:

$$\begin{aligned} u^{*k}_{i\to j} &= \alpha u^k_{i\to j} \\ u^k_{i\to s} &= (1-\alpha) u^k_{i\to j} \end{aligned} \tag{1}$$

   where $0 \le \alpha < 0.5$ is a system parameter, and it determines the amount of unhappiness passed to a service process when an indirect request is sent to such a process. Its exact value should be determined based on experiments during a specific implementation.

4. The new unhappiness value $u^{*k}_{i\to j}$ for P_j does not change while process P_j is blocked waiting for a service from other processes. But $u^k_{i\to s}$ will increase by time and in general follows the same rules for unhappiness computation.
5. When the service process P_s finishes its processing and returns a response to P_j, effectively unblocking P_j by giving the requested service to it, the value of $u^k_{i\to s}$ is passed back to P_j and is added to its previous unhappiness value. We call this new unhappiness value $u^{**k}_{i\to j}$. At this time the unhappiness value assigned to service process is reset to zero:

$$\begin{aligned} u^{**k}_{i\to j} &= u^{*k}_{i\to j} + u^k_{i\to s} \\ u^k_{i\to s} &= 0 \end{aligned} \tag{2}$$

We can compute the total unhappiness for a request, a customer (U^{c_i}) or the whole system (U). The total unhappiness for a request $R^k_{i\to j}$ is computed as the following summation which indicates the total unhappiness that customer C_i experiences as a result of sending request $R^k_{i\to j}$ to process P_j and all other delays that are caused as process P_j waits for services from other processes.

$$U^{R_i^k} = W(c_i) w_i^k \sum_{j=0}^{N-1} u^k_{i\to j} \tag{3}$$

The total unhappiness for customer C_i is computed using Equation 4. In this equation R is the total number of requests sent by C_i and N is the total number of processes in the system:

$$U^{c_i} = W(c_i) \sum_{j=0}^{N-1} \sum_{k=0}^{R-1} w_i^k u^k_{i\to j} \tag{4}$$

The system unhappiness due to all requests from all customers is computed using Equation 5:

$$U = \sum_{i=0}^{M-1} W(c_i) \sum_{j=0}^{N-1} \sum_{k=0}^{R-1} w_i^k u^k_{i\to j} \tag{5}$$

The objective of the scheduling algorithm should be to minimize the system's unhappiness U at any time.

3 Unhappiness Values in different Scheduling Algorithms

In order to find out how some of the scheduling algorithms perform under the customer appeasement model, we compute the unhappiness value that they cause for a request sent by a customer to a system in two specific scenarios. We compute the unhappiness value for the following schedulers, and simplify final values as much as possible so that the results are comparable.

3.1 Round-Robin Scheduling

The Round-Robin (RR) Scheduler is a simple preemptive scheduling algorithm which was used in time sharing systems [15]. It is still used as part of some modern scheduling algorithms, for example it is part of Linux real time (RT) scheduling class. Round-Robin scheduler gives each process a time slice or time quantum q, if a process releases the CPU before q is finished, then the scheduler runs the next process in the ready queue. If a process needs more time and finishes its time quantum, then the scheduler preempts the process, inserts the process at the tail of the ready queue, and schedules the next process from the head of the ready queue. This new running process also receives a time quantum q.

Now we consider a simple case and compute the minimum and a typical unhappiness values caused by a request in a system with RR scheduler. The minimum and typical unhappiness values might happen in the best case and a typical case scenarios respectively. We assume that there are $N-1$ running processes in the run queue. We assume that there is a process P_j in the sleeping state waiting to receive a request. There is only one customer, and the system parameter α is set to zero ($\alpha = 0$). This customer sends a request to the sleeping process P_j and waits for the response. Now the OS wakes up process P_j and inserts it to the end of the ready queue. Assume that the $N-1$ running processes stay running all the time, and they use all of their time quanta , so P_j waits in the ready queue for $q(N-1)$ seconds before it runs on the CPU. If it can finish processing and return a response to the customer during its first time slice q, then $q(N-1)$ is the amount of unhappiness during this transaction. If it can't return a response to the customer during this time and needs a total of Z time quanta to finalize this transaction and return a response to the customer, then the maximum unhappiness will be:

$$U^R = Zq(N-1) - (Z-1)q = q(Z(N-2)+1) \tag{6}$$

In practice it is possible that process P_j blocks and waits for services from other processes. Assume that it waits for a service from process P_s after Z_1 time quanta, and P_s needs Z_2 time quanta to process P_j's request and return a response. P_j may also need another Z_3 time quanta to return a response to the customer. We assume that P_s is in the sleeping state prior to receiving a request from P_j, that means, there are always N running processes, because when P_j blocks and sleeps, P_s wakes up and is in the running state. Then the amount of unhappiness experienced by the customer due to the request R will be:

$$\begin{aligned} U^R &= q(Z_1(N-2)+1) + q(Z_2(N-2)+1) + q(Z_3(N-2)+1) \\ &= q((Z_1+Z_2+Z_3)(N-2)+3) \end{aligned} \tag{7}$$

Here the unhappiness value consists of three terms. The first term is the aggregated unhappiness caused by delays in the execution of P_j at the time it blocks waiting on P_s. The second term is the amount of unhappiness caused by delays during the execution of P_s, and the last term of the unhappiness value reflects the delays of running P_j after it receives the response from P_s until it sends the final response to the customer. So the best case and a typical case scenarios with a Round-Robin scheduler results in the following minimum and typical unhappiness values:

$$U^R_{min} = q(Z(N-2)+1) \tag{8}$$
$$U^R = q((Z_1+Z_2+Z_3)(N-2)+3) \tag{9}$$

Note that the unhappiness value related to running the service process P_s is set to zero once it sends the response back to P_j.

3.2 Multilevel Feedback Queues

In this subsection we perform the same analysis for a basic multilevel feedback queue scheduling. Many UNIX OSs such as FreeBSD [9] utilize some form of a multilevel feedback queue scheduler. Windows also has a multilevel feed back queue scheduler [13]. As another example the $O(1)$ or Vanilla scheduler in Linux kernels before 2.6.23 is in fact a multilevel feedback queue with many heuristics involved in moving tasks between diffrent queues and detecting interactive processes [11].

Assume a basic multilevel feedback queue scheduling algorithm with m queues called Q_0 to Q_{m-1}. The processes in each queue can run on the CPU for a multiple of time quantum q. The amount of time for Q_i is computed as $2^i q$. Each process is first placed in Q_0. After it receives its CPU share in Q_0, if it needs more time it is then placed in the next queue Q_1 and so on. Each queue has an absolute priority relative to the next queue, meaning that processes in Q_{i+1} do not execute until all processes in Q_i receive their CPU share and Q_i becomes empty. So in this algorithm, processes which need more CPU time, lose their priority as time passes but receive a,larger time quantum when they run on the CPU.

Now assume an interactive process wakes up and receives a request from a customer. Also assume that there are a total of a_i processes in Q_i. Assume that the interactive process needs Zq processing time to finish processing and return a response to the customer, and no other processes will enter the running queues during this time. This means that the interactive process is inserted to the end of Q_0, receives its CPU time after waiting for other processes in this queue, then it is pushed to the next queue and so on. Assume Q_x is where it receives the final amount of CPU time that it needs to finish processing the request and return a response to the customer. The amount of unhappiness that the customer experiences is computed as:

$$\begin{aligned} U^R &= (a_0-1)q + 2(a_1-1)q + ... + 2^x(a_x-1)q - (q + 2q + ... + 2^{x-1}q) \\ &= q(\sum_{i=0}^{x} 2^i(a_i-1) - \sum_{i=0}^{x-1} 2^i) \end{aligned} \tag{10}$$

In this equation the first summation indicates the amount of delays that the interactive process encounters waiting in ready queues, and the second summation indicates the amount of CPU time it has received. We can compute x by solving this equation $\sum_{i=0}^{x} 2^i = Z$ and then simplifying the minimum unhappiness value as follows:

$$x = \log_2(Z+1) - 1 \tag{11}$$

$$U^R_{min} = q(\sum_{i=0}^{\log_2(Z+1)-1} 2^i(a_i-1) - 2^{\log_2(Z+1)-1}) \tag{12}$$

We can also examine a more complicated scenario as in the previous subsection. Assume that P_j wakes up and receives a request. It then spends Z_1 second to partially process this request and send a request to a service process P_s. Now P_s needs Z_2 seconds to provide the service to P_j. After P_j receives the service, it needs another Z_3 seconds to finalize its processing and return a request to the customer. Please note that in this scheduler each time a process wakes up, it is inserted to the end of Q_0. We compute unhappiness values assuming that the system parameter α is set to zero ($\alpha = 0$) see Section 2:

$$U^R = q(\sum_{i=0}^{\log_2(Z_1+1)-1} 2^i(a_i - 1) - 2^{\log_2(Z_1+1)-1}) + q(\sum_{i=0}^{\log_2(Z_2+1)-1} 2^i(a_i - 1) - 2^{\log_2(Z_2+1)-1})$$
$$+ q(\sum_{i=0}^{\log_2(Z_3+1)-1} 2^i(a_i - 1) - 2^{\log_2(Z_3+1)-1}) \tag{13}$$

Please note that the total unhappiness value consists of three terms. The first term is the result of delays during the first part of processing the request by P_j. The second term is caused when the service process P_s is in the run queue, and the last term is associated with the waiting when P_j prepares the final response for the customer. This scenario can be a typical situation while it is possible to have even more complex cases where P_j may need more services from P_s or other service processes. This leads to higher unhappiness values.

3.3 Linux CFS Scheduler

The Completely Fair Scheduler or CFS for short [10, 12] was introduced in Linux kernel version 2.6.23. As its developer explains [10, 12], "it is designed to basically model an ideal, precise multi-tasking CPU on real hardware".

It allways tries to share the CPU fairly between the current processes in the run queue. In other words if there are N processes in the run queue, it promises to run each process with $\frac{1}{N}$ of the CPU power. In order to achieve this goal, CFS scheduler keeps track of a variable called virtual run time (*vruntime*) for each task. It is a weighted run time for each task. CFS uses an R-B tree to choose the next task to run on the CPU. It simply chooses the left most task in the tree which has the lowest *vruntime* value. If a new process enters the run queue, CFS manipulates its *vruntime* value such that the new arriving process goes to the right of the R-B tree. This is to make sure that it can keep its promised wait time to the current running processes. CFS also gives an advantage to the sleeping processes. If a process sleeps less than a threshold time interval then CFS changes its *vruntime* value such that it goes to the left most position in the R-B tree when it wakes up. Assuming that an interactive task is a short sleeper, this will lead to better response times for interactive tasks. CFS does not have a fixed time slice. At the time it runs the next task it gives the task a time slice which is computed as follows:

$$Timeslice = q(N) = \frac{sch\_lat}{N} \tag{14}$$

In this equation *sch_lat* is a CFS constant value and N is the number of tasks in the run queue. CFS stretches this time slice if the number of running tasks increases beyond a system threshold. $q(N)$ is the basic time slice in CFS if process weights and nice values are ignored.

Another intresting property of CFS scheduler is the way nice values work. Nice values change the weight of a task, which means that they change the *vruntime* of a task. If task weights and nice values are considered then CFS changes the CPU share of each task based on the following relations:

$$\begin{aligned} q(P_j, N) &= \frac{w_j sch\_lat}{\sum_{i=1}^{N} w_i} \\ wnice_0 &= 1024 \\ wnice_{i-1} &= 1.25 wnice_i \end{aligned} \tag{15}$$

w_i in equation 15 is the weight of P_j assigned by CFS, and changes directly based on the P_j's nice value. For example if there are two tasks A and B running on a single CPU machine and both have a default nice value of 0 then each receives 50% of the total CPU time. If task A's nice value is changed to -1 then it receives 55% of the CPU time and task B receives 45% of the CPU power.

Now we compute the unhappiness values experienced by a customer in a system with CFS scheduler. Assuming that all running processes have their default nice value of zero, we have:

$$q(P_j, N) = \frac{1024 \quad sch\_lat}{1024 \quad N} = \frac{sch\_lat}{N} = q(n) \tag{16}$$

Assume that there are $N - 1$ running processes in the run queue and an interactive task P_j is sleeping. Assume a customer sends request R to P_j. P_j wakes up and enters the run queue. Now we compute the minimum unhappiness that may be experienced by the customer. In the best case scenario, it is possible that CFS changes the *vruntime* of P_j such that it is placed in the left most leaf of the R-B tree and it then may preempt the current running process. Please note that now the number of running processes is N. Assume that P_j needs τ seconds to finish processing the request. If $\tau \leq q(N)$ then P_j can finish processing the request without interruption and returns a response to the customer and the total unhappiness value is zero.

$$U^R_{min} = 0 \tag{17}$$

Now we assess a typical scenario where $\tau > q$ and the interactive task blocks to receive a service from service task P_s. We assume that the system parameter $\alpha = 0$, and when P_j blocks, it transfers all of its unhappiness to P_s. At this time P_s enters the run queue so the total number of running jobs does not change. Additionally we also assume that no other task enters or leaves the run queue while the request R is being serviced. So the total number of running tasks is always N. Assume that P_j needs τ_1 seconds for processing the request R before blocking and requesting service from P_s, and P_s needs τ_2 seconds to return the requested service to P_j, and P_j needs another τ_3 seconds to return a response to the customer. To simplify the computation we use $\tau = \tau_1 + \tau_2 + \tau_3$. The total unhappiness experienced by customer due to this request is:

$$\begin{aligned} U^R &= (\frac{\tau_1}{q(N)} - 1)(N-1) - \tau_1 + (\frac{\tau_2}{q(N)} - 1)(N-1) - \tau_2 + (\frac{\tau_3}{q(N)} - 1)(N-1) - \tau_3 \\ &= (N-1)(\frac{1}{q(N)}(\tau_1 + \tau_2 + \tau_3) - 3) - (\tau_1 + \tau_2 + \tau_3) \\ &= (N-1)(\frac{1}{q(N)}\tau - 3) - \tau \end{aligned} \tag{18}$$

4 Scheduling Based on Unhappiness Values

Up to this point, we have developed a model which can give us an indication of how an OS performs regarding the requests it receives from different customers. Interstingly, the way we have defined unhappiness, and compute its value, and the way it is inherited by processes, highlights the dependency between processes which are responsible for a particular request. We can look at this as a way of coloring a process dependency subgraph which is involved in responding to a particular request, and finding the process which creates the most unhappiness value at the current time. The objective is to minimize system unhappiness. The very first idea to achieve this is to find the request which creates the maximum aggregated unhappiness and allocate the CPU to the processes responsible for serving that request. In theory we can achieve this by performing the following steps.

1. We assume that, there are two queues in the system. One for unhappy processes which is called Q_0 and the second queue for regular processes which is called Q_1.
2. All processes with nonzero unhappiness values are placed in Q_0 and processes with zero unhappiness values are placed in Q_1.
3. Q_0 has an absolute higher priority than Q_1, so there should be no processes in Q_0 before processes in Q_1 can be executed.
4. In order to determine which process to execute next from Q_0, the scheduler first computes the unhappiness values for all pending requests in the system.
5. Based on the requests' unhappiness values computed in the previous step, the scheduler chooses the request with the highest unhappiness value.
6. The scheduler then, checks all processes responsible for that request. In other words it checks the dependency subgraph of the processes that are servicing that particular request and chooses the process with the highest unhappiness value to run on the CPU.
7. Processes in Q_1 are executed based on a regular system scheduling algorithm for example Linux CFS scheduling algorithm.
8. When a new process is created it should be given a nonzero unhappiness value so that it has a chance to start faster, then if it does not serve requests, it is moved to Q_1.

5 Request Based Priority Elevation for CFS

Unfortunately, implementing the simplest version of the proposed algorithm in Section 4 requires that the scheduler detects all requests to all processes, and also requires detecting responses from processes to customers. At present detecting a response from a process to a particular request seems to be impossible without process cooperation. As a result of facing these difficulties, and in order to create a proof of concept implementation, we decide to take a simplified approach. We begun by observing a typical Linux desktop which was also configured as a small web server. This system had a Linux kernel version 2.6.31 and hence it uses CFS scheduler. We used SystemTap [6] and strace [5] to trace system calls and interprocess communications. We observed that most of the requests from a desktop user is passed to processes through UNIX sockets. Desktop components also mostly use UNIX sockets to communicate with each other [2, 3]. Requests from remote customers also enter the system through network sockets. Based on these observations we propose a simple priority elevation technique in Linux kernels with CFS scheduler to approximately minimize system unhappiness as defined in Section 2.

1. Assume that each process receives the incoming requests through a non-zero socket read.

2. Since a pending request increases system unhappiness, whenever a process receives a request (non-zero socket read), scheduler should increase its priority or CPU share.

3. The process should be able to maintain it's elevated priority until it sends a response to the customer. But, as we can't detect the exact time when it sends a response back to the customer, scheduler decays the elevated priority in time.

4. In order to minimize interaction with regular system scheduling, we change the amount of priority elevation and decaying speed based on the system load. As the system load increases, an eligible process receives higher priority and retains this higher priority for a longer time. This is based on the fact that when a Linux OS with CFS scheduler has a higher load, each process receives a smaller share of the CPU time [10, 12]. So, in order for a given process that receives a request to be able to respod to the request as if there is little or no load on the system, it should receive a larger share of CPU time relative to other processes. By increasing its priority more aggressively under heavier loads, we allocate more CPU time to such a process, as a result, it has a better chance to finish its computation and return a respond to the request in a shorter time interval.

In the rest of this paper we refer to this method by its abbreviation CFS/RBPE.

5.1 Unhappiness Values for CFS/RBPE

In this subsection we compute theoretical unhappiness values for the priority elevation technique as we did for other schedulers in Section 3. Again we consider two scenarios under this scheme and compute the amount of unhappiness observed by the customer. The assumptions are mostly the same as what we assume in Section 3.3. There are $N-1$ tasks in the run queue of a single processor machine. Task P_j is an interactive job which is sleeping. A customer sends a request R to P_j. It wakes up and is inserted in the run queue. Assume that priority elevation mechanism detects the request and uses negative nice value $-nice$, such that $-15 \leq -nice < 0$ to increase P_j's priority. Assume that elevated priority decays with a speed of 1 nice level per each d seconds. P_j needs τ seconds to finish computation and return a response to the customer, and $d < \tau$. As we explained in Section 3.3, each negative nice level increases the task's weight to 1.25 times its previous value. So if there are N tasks including P_j in the run queue and P_j has a negative nice value $-nice$ then the following relations hold:

$$\begin{aligned} q(P_j, N) &= \frac{1.25^{nice} \quad sch\_lat}{N-1+1.25^{nice}} \\ q(P_{i \neq j}) &= \frac{sch\_lat}{N-1+1.25^{nice}} \end{aligned} \tag{19}$$

Assume $\tau \leqslant \frac{1.25^{nice} \quad sch\_lat}{N-1+1.25^{nice}}$ then as CFS enqueues an interactive task into the left of R-B tree, it is possible that P_j finishes computation and returns a response to the customer within this time period. This means that, it is possible that the customer observes zero unhappiness.

$$U^R_{min} = 0 \tag{20}$$

Now, let us consider a typical scenario where $\tau \geqslant \frac{1.25^{nice} \quad sch\_lat}{N-1+1.25^{nice}} = q$ and P_j also needs a service from process P_s. We also assume that P_j first requires $\tau_1 \approx Z_1 q$ seconds before it blocks for service from P_s. P_s requires $\tau_2 \approx Z_2 q$ to provide service to P_j, and eventually P_j requires another $\tau_3 \approx Z_3 q$ to return a response to the customer. Also assume that $d \leqslant q$, so P_j and P_s nice levels increase almost every time it is rescheduled after it is set to $-nice$ by the scheduler. For simplicity we also assume that $Z_1, Z_2, Z_3 \leqslant nice + 1$. Based on these assumptions the amount of unhappiness observed by customer is:

$$\begin{aligned} U^R = &\left(\sum_{i=0}^{Z_1-2} \frac{(N-1)sch\_lat}{N-1+1.25^{(nice-i)}} - \sum_{i=1}^{Z_1-1} \frac{1.25^{(nice-i)}}{N-1+1.25^{(nice-i)}}\right) \\ &+ \left(\sum_{i=0}^{Z_2-2} \frac{(N-1)sch\_lat}{N-1+1.25^{(nice-i)}} - \sum_{i=1}^{Z_2-1} \frac{1.25^{(nice-i)}}{N-1+1.25^{(nice-i)}}\right) \\ &+ \left(\sum_{i=0}^{Z_3-2} \frac{(N-1)sch\_lat}{N-1+1.25^{(nice-i)}} - \sum_{i=1}^{Z_3-1} \frac{1.25^{(nice-i)}}{N-1+1.25^{(nice-i)}}\right) \end{aligned} \tag{21}$$

Assuming the total time needed to return a response to the customer is τ, then we have $\tau = \tau_1 + \tau_2 + \tau_3$. Note that the CPU time that P_j and P_s receive is approximately the total execution time that they need to return a response to the customer request, so:

$$\tau = \sum_{i=1}^{Z_1-1} \frac{1.25^{(nice-i)}}{N-1+1.25^{(nice-i)}} + \sum_{i=1}^{Z_2-1} \frac{1.25^{(nice-i)}}{N-1+1.25^{(nice-i)}} + \sum_{i=1}^{Z_3-1} \frac{1.25^{(nice-i)}}{N-1+1.25^{(nice-i)}} \tag{22}$$

And we can write U^R as the following:

$$U^R = \sum_{i=0}^{Z_1-2} \frac{(N-1)sch\_lat}{N-1+1.25^{(nice-i)}} + \sum_{i=0}^{Z_2-2} \frac{(N-1)sch\_lat}{N-1+1.25^{(nice-i)}} + \sum_{i=0}^{Z_3-2} \frac{(N-1)sch\_lat}{N-1+1.25^{(nice-i)}} - \tau \tag{23}$$

Please note that the above calculations are based on the assumption that all communications/requests between processes are performed using network or UNIX sockets. Based on our observation this is a valid assumption for most Linux desktop applications/components. Another fact is that most transactions need more than just one read operation. For example a simple click on a link in a web browser causes many UNIX socket read system calls both in the web browser and the X server before a new page is displayed on the screen. This in effect causes the active processes in the transaction (in this case web browser and X server) to receive the $-nice$ value multiple times. It means that in practice, they have higher priority for a longer period of time than what we compute here.

5.2 CFS vs CFS/RBPE

In this subsection we compare and discuss unhappiness computations for CFS scheduler and CFS with RBPE. As we see in sections 3.3 and 5.1 the minimum unhappiness values for the best scenarios in both cases are zero. So at the very minimum we can see that in theory our proposed RBPE scheme does not make the situation worse. We can note that there are two conditions for the best case scenarios with the resulting zero unhappiness. These conditions are:

$$\tau \leqslant \frac{sch\_lat}{N} \qquad For\ CFS. \tag{24}$$

$$\tau \leqslant \frac{1.25^{nice}\ sch\_lat}{N-1+1.25^{nice}} \qquad For\ CFS/RBPE. \tag{25}$$

As $\frac{1.25^{nice}}{N-1+1.25^{nice}} \geqslant \frac{1}{N}$ clearly under CFS/RBPE, P_j has more time to respond to the request before it is preempted than under CFS. So, there is higher possibility under CFS/RBPE that a request being responded without encountering any unhappiness.

For typical scenarios in both cases again we have:

$$U^R = (N-1)(\frac{\tau}{sch\_lat} - 3) - \tau \qquad For\ CFSand \tag{26}$$

$$U^R = (N-1)sch\_lat(\sum_{i=0}^{Z_1-2} \frac{1}{N-1+1.25^{(nice-i)}} + \sum_{i=0}^{Z_2-2} \frac{1}{N-1+1.25^{(nice-i)}} + \sum_{i=0}^{Z_3-2} \frac{1}{N-1+1.25^{(nice-i)}}) - \tau \qquad For\ CFS/RBPE \tag{27}$$

In the CFS/RBPE case, as a result of higher priority for P_j and P_s, other tasks have less CPU time, so the resulting U^R value is less than that of CFS case.

6 Implementation

While we were using SystemTap [6] to observe process/OS interactions and behaviors, we found it extremely powerful to write simple scripts which can be used with different kernel versions without almost any modification. So, in order to implement a simple Request Based Priority Elevation (RBPE) mechanism as a proof of concept for our customer appeasement, we decided to use SystemTap in its Guru mode. When used in this mode, SystemTap enables parsing of expert-level constructs like embedded C. So it basically enables us to write C code and insert it into the kernel as a kernel module, effectively modifying a running kernel without directly modifying kernel source code or recompiling it.

We use systemTap to create a list of process (PIDs) that recently have called a socket related receive/read system call with nonzero return value. Assuming this call means that the associated process has received either a direct or indirect request from a customer, we increase its priority. The exact negative nice value used to increase the process priority depends on the system load. The higher the system load the lower the negative nice value. The applied negative nice value along with a time stamp is saved in a list with the associated process PID. We call this list elevated priority process list (EPPL). There is a time delay after that we increase the negative nice value of the processes which are in the EPPL, effectively reducing their priority. The exact value of this time delay also depends on the system load. During our process behavior observation period, we noticed that, the majority of processes waiting for an input, use the poll system call periodically. We changed the poll system call and use it as a point to check the current system status and update the state of the processes which are in the elevated priority list. Each time a poll is called, we check EPPL and increase the nice value by one for each process that has passed its delay time. We then enter the new nice value with a new time stamp into the elevated process list. For each process in the list, this action will continue until the nice value becomes zero, at that point the process is deleted from the list. We adjust the values of initial nice values and delay time, based on the system load during poll system calls. If system load is very low, the RBPE mechanism does not interfere with the regular CFS scheduling decisions, but, as the system load inceases it interferes aggressively, as mentioned earlier. Table 6 shows the initial nice vales and delay times at different system loads in the current version (0.5) of our script.

In our implementation we discriminate local and remote users by giving higher priority to processes receiving requests through UNIX sockets relative to those processes receiving requests through network sockets. This is implemented by using two initial negative nice values. One with a lower value for processes that use local UNIX sockets and one with higher value for processes that use network sockets to receive requests. All of these values are presented in Table 6.

Although this method of implementation might have a higher overhead, but as we see in Section 7, it has very promising results for real world applications.

avenrun[0] $\leq$	*-nice1* (UNIX sockets)	*-nice2* (Network sockets)	Time Delay (ms)
1600	0	0	0
3000	-1	0	200
5000	-2	-1	300
8000	-4	-2	400
12000	-6	-3	500
16000	-7	-4	600
$>$ 16000	-15	-5	600

Table 1: RBPE script, nice and time delay values. avenrun[0] is a kernel variable which represents system's 1 minute load.

7 Experiments

To test our CFS/RBPE technique, we performed multiple tests. As we intend to show that our scheduling paradigm does not focus on interactive applications, we have performed server based performance tests as well as regular interactivity/multimedia tests. Server based tests include Apache web server [1] performance test and Mysql data base server [4] performance tests. We choose these two servers as they are very popular. In fact many Linux based servers use Apache, Mysql and PHP to support different weblog, wiki or multimedia hosting services.

7.1 Hardware/Software Set up

All tests are performed on an IBM (R) IntelliStation M Pro with Intel *P4* 2.8GHz CPU and 1GB of RAM running Fedora 12 with Kernel 2.6.32.

In order to simulate different background system loads we compile Linux kernel and use different *-j* values with the *make* command to initiate different parallel compilations.

7.2 Apache Web Server Response Test

As a first test to measure server based application performance we measure the response time of Apache web server under different system loads. The web server hosts a directory structure of files. We use a second machine and *wget* command to download the complete directory structure from the web server. We use shell *time* command to measure wall clock download times under different simulated load conditions. For each simulated load value we repeat the experiment 3 times and compute the average download time for that system load. As we see in Figure 7.2, both CFS and CFS/RBPE have the same response time for *-j* values up to 2 after that point CFS/RBPE has consistently lower response times. When *-j* value equal to 30 is used Apache web server is almost 1.5 times slower when it is run under CFS than under CFS/RBPE.

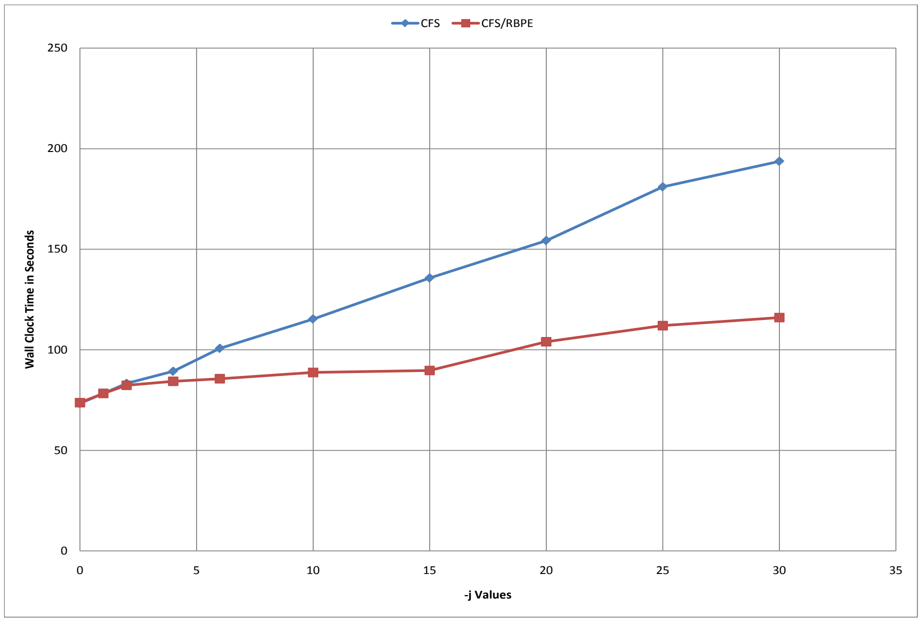

Figure 1: Download times in Seconds from an Apache web server under different simulated load values.

7.3 SysBench Mysql Data Base Test

As a second server based software, we test Mysql response times under different background load conditions. We use Stench [16] to evaluate and compare Mysql data base performance under Linux CFS [10, 12] and CFS/RBPE scheduling policy. Stench is a multi-threaded benchmark tool which can evaluate OS parameters important for a system running on a data base server under heavy load. We use Stench default parameters for its OLTP complex configuration. During each experiment Stench executes many read and write transactions on the Mysql data base server, and then gives the average transaction time. Each experiment runs for 120 to 250 seconds under different simulated load conditions. We change experiment times in order to have around 3000 transactions per each experiment. The reason is that, when system load increases the total number of transactions in a fixed time interval decreases. So we increase the experiment time to have almost equal number of transactions for each experiment. This gives us a better average transaction time in all experiments. Figure 7.3 shows the average transaction time in milliseconds under different simulated load conditions. Load conditions were simulated by running a Linux kernel compilation job with different *-j* options to the *make* command as in the previous test in Section 7.2.

As we see in Figure 7.3, the average Mysql tarnsaction time under CFS/RBPE increases first as *-j* value increases to 1 (meaning from no load to one parallel kernel compilation)

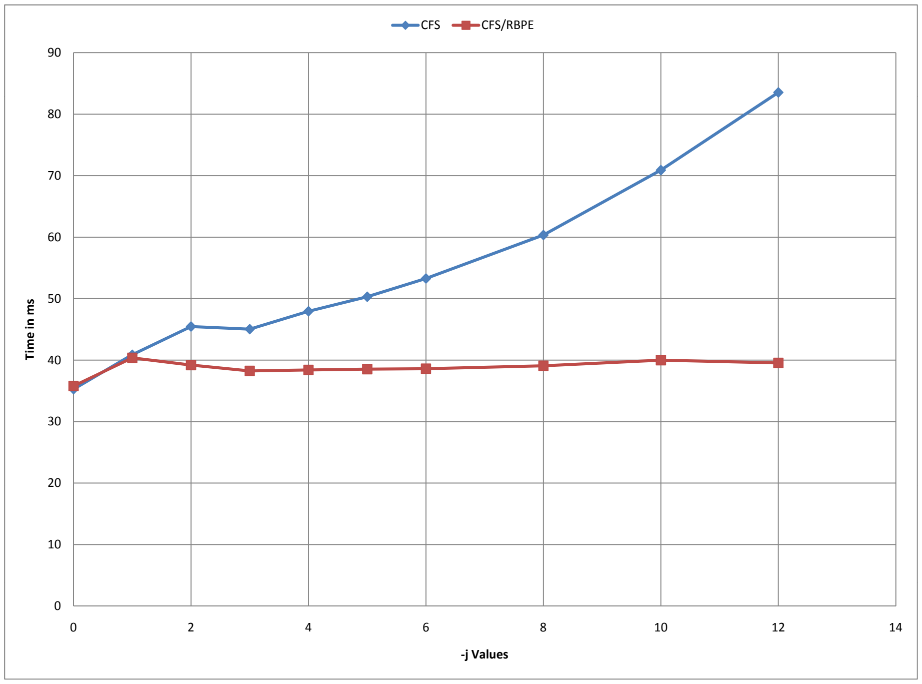

Figure 2: It shows the average transaction times in milliseconds under different simulated load conditions. The X values are *-j* values passed to the make command to specify number of parallel compilations to initiate.

but, then it decreases when parallel compilation increased to two and three. This is because RBPE does not interfere with the usual CFS scheduling when system load is low. When the system load increases RBPE is involved and as we see in Figure 7.3, it boosts Mysql performance so that its average transaction time is almost constant near 40 milliseconds. In contrast, under CFS scheduling, Mysql average transaction time increases as system load increases. When we use a *-j* value of 12, the average transaction time reaches 85 milliseconds which is more than two times that of CFS/RBPE.

The results of the experiments in this section and the previous section indicate that the proposed scheduling paradigm in this paper is not a method to just boost desktop interactive applications response time. This mechanism can boost the performance of any request/response based transaction in the system.

7.4 Interactivity/Multimedia Test

In order to test and compare the performance of interactive/multimedia applications under CFS and CFS/RBPE schedulers, we use *mplayer* in benchmark mode. In this mode *mplayer*

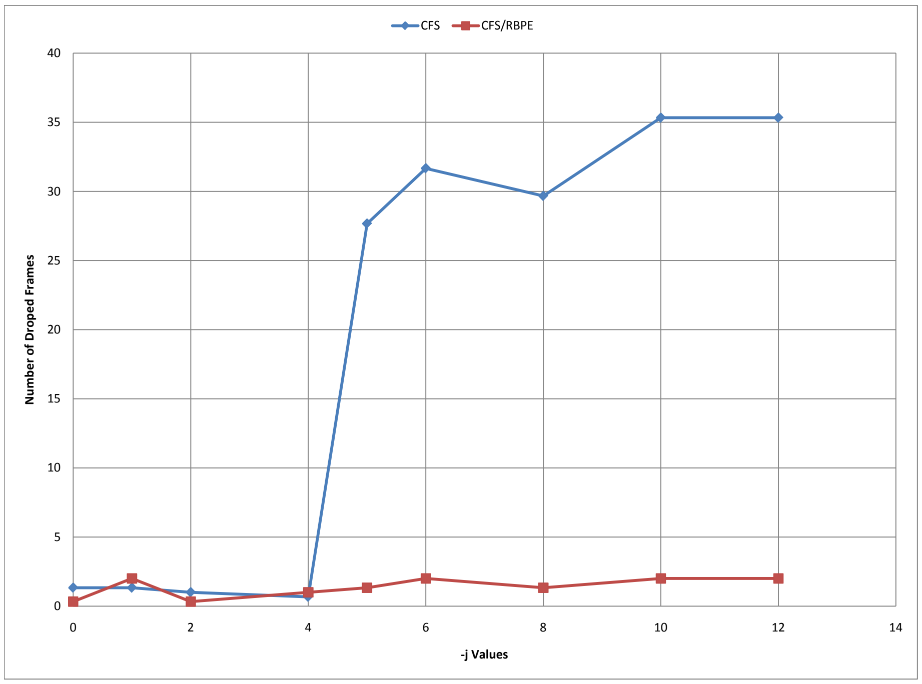

Figure 3: This figure compares the frame rate drop when mplayer is playing an mpeg movie clip and a simulated background load was increasing.

prints out the number of dropped frames and average frame rate after it finishes playing a multimedia file. We use a short mpeg clip of size 352 x 288 pixels, which runs for about 150 seconds. The clip frame rate is 25 frames per second. Again we simulate the system load with parallel compilation of Linux kernel and use *make -j* with different values for *-j* to control the number of parallel *makes*.

For each *-j* value the experiment is repeated three times and the average value of dropped frames is depicted in Figure 7.4. As we see in this graph under CFS/RBPE the frame drop is almost zero for all load values up to *-j 12*. Under CFS the number of frame drops significantly increases after *-j 4*.

We also depict the average frame rate of *mplayer* for CFS and CFS/RBPE under different simulated load levels. As we see in Figure 7.4, *mplayer* frame rate drops to almost 8 frame per second under CFS when 12 parallel compilation is running, while at the same load level CFS with RBPE shows almost no frame rate reduction.

This experiment shows the effectiveness of CFS/RBPE on a typical multimedia or streaming application. As Figures 7.4 and 7.4 indicate, basically the movie is not viewable when *-j* value reaches 5 on our system with CFS scheduling. In contrast a viewer can still enjoy watching a movie on the same system if CFS/RBPE scheduling is used even if *make -j* with

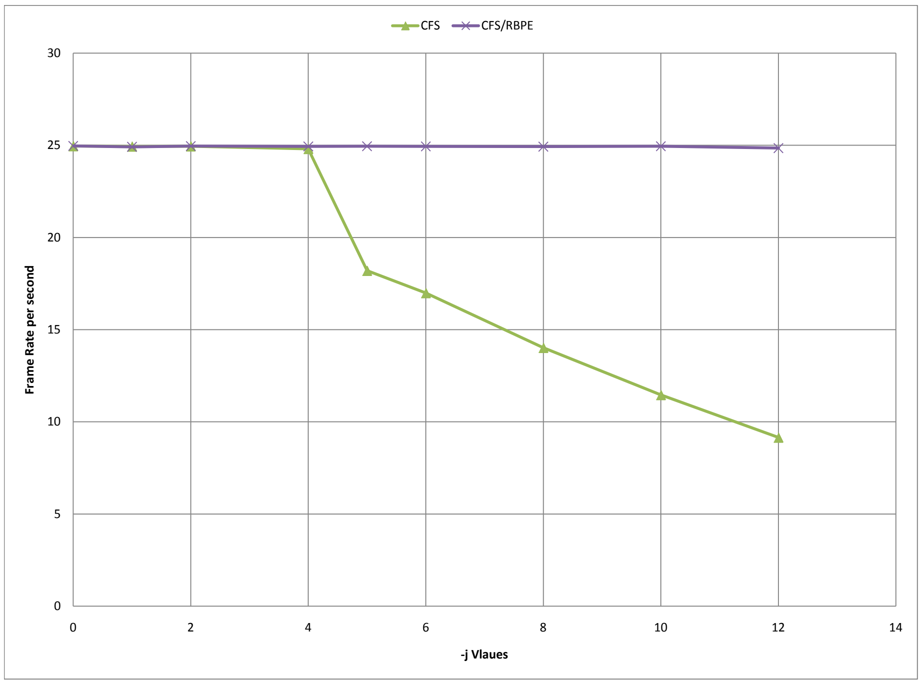

Figure 4: Demonstrates frame rate change due to system load under CFS scheduler and CFS/RBPE.

value of 12 is used for compiling a linux kernel at the same time.

8 Concluding Remarks

In this work we introduce a new policy for CPU scheduling. This policy is based on tracking requests sent by customers to different processes and their response to the requests. We assume that a computer system should allocate its resources such that the customers do not experience excessive delays. We have defined a model which can be used to analyze and compare different scheduling algorithms based on this assumption. This model considers delays resulted from processes dependencies. When a requests arrives at the system, one or more processes are responsible to execute the request and return a response to the customer. We detect the request and the processes which are responding to the request by tracking interprocess communications. We have a minimal implementation on top of Linux CFS scheduler [10, 12] as a proof of concept which increases the priority of the processes involved in a response to a customer request. Our experiments show that this mechanism is not only effective for improving interactive/desktop applications performance under heavy system load, it is also effective for improving server applications under heavy background load. Experiments with Apache web server and Mysql data base server in Sections 7.2 and 7.3, show significant performance boost for these server applications under heavy background load. A server background load may be the result of disk indexing, data base indexing, log rotation, log analysis, etc.

One of our goals is to make the implementation simple, easily portable to different Linux kernels and distributions, and easy to use for a novice user. In order to achieve this, we use SystemTap [6]. SystemTap has made our implementation a simple script which can be run on any SystemTap equipped distribution with a compatible kernel without the need to recompile and install a new kernel. There has been a debate and disagreement among kernel development community on whether to support SystemTap or not [7]. If the support for SystemTap is dropped by its main developers then as far as we know there is no alternative for a fast and easy to use implementation as we have done in this work.

Some of the possible future works are:

1. Integration of the implementation with the Linux kernel instead of using SystemTap.
2. Extending the proposed CPU scheduling paradigm to disk scheduling in the sense that higher priority process also get higher priority disk access.
3. Studing the effects of adding other interprocess communications like pipes to the implementation.
4. Adding other mechanisms to detect arrival of a request to the system.
5. Finding ways to give different weights to different requests from a specific customer.
6. Studing the usage of proposed mechanism in managing resources used by different virtual machines on one real machine.

References

[1] Apache web site. http://www.apache.org/.

[2] Dbus specifications. http://dbus.freedesktop.org/doc/dbus-specification.html.

[3] Kde developer's web site. http://developer.kde.org/documentation/other/dcop.html.

[4] Mysql web site. http://www.mysql.com/.

[5] Strace is a linux system cammnad. http://linux.die.net/man/1/strace.

[6] Systemtap web site. http://sourceware.org/systemtap/.

[7] Jake Edge. Back to the drawing board for utrace? http://lwn.net/Articles/371210/.

[8] Yoav Etsion, Dan Tsafrir, and Dror G. Feitelson. Process prioritization using output production: Scheduling for multimedia. *ACM Trans. Multimedia Comput. Commun. Appl.*, 2(4):318–342, 2006.

[9] Marshall Kirk McKusick and George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System.* Pearson Education, 2004.

[10] I. Molnar. Linux cfs scheduler. http://www.kerneltrap.org/node/11737.

[11] I. Molnar and C. Kolivas. Interactivity in linux kernel 2.6. http://www.kerneltrap.org/node/780.

[12] Ingo Molnar. A description of CFS design. http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt.

[13] Mark Russinovich and David A. Solomon. *Windows Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition.* Microsoft Press, 2009.

[14] Amit Singh. *Mac OS X Internals.* Addison-Wesley Professional, 2006.

[15] Andrew S Tanenbaum and Albert S Woodhull. *Operating Systems Design and Implementation (3rd Edition).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[16] Sysbench team. Sysbench documentation. http://sysbench.sourceforge.net/docs/.

[17] Le Yan, Lin Zhong, and Niraj K. Jha. Towards a responsive, yet power-efficient, operating system: A holistic approach. *Modeling, Analysis, and Simulation of Computer Systems, International Symposium on*, 0:249–257, 2005.

[18] Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Redline: First class support for interactivity in commodity operating systems. In Richard Draves and Robbert van Renesse, editors, *OSDI*, pages 73–86. USENIX Association, 2008.

[19] Haoqiang Zheng and Jason Nieh. Swap: a scheduler with automatic process dependency detection. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 14–14, Berkeley, CA, USA, 2004. USENIX Association.

[20] Haoqiang Zheng and Jason Nieh. Rsio: automatic user interaction detection and scheduling. In *SIGMETRICS*, pages 263–274, 2010.