Human-Centered Scheduling of Interactive and Multimedia Applications on a Loaded Desktop

Abstract

While modern desktop workloads include a substantial multimedia component, virtually no contemporary general purpose operating system provides adequate support for multimedia applications when executed under loaded conditions. Trying to play a DVD movie or participating in a role playing game with significant graphical requirements while running demanding assignments in the background (such as compiling the Linux kernel or contributing to the SETI@home effort) will usually result in poor graphical quality. This happens because general-purpose schedulers prioritize processes mainly based on their CPU consumption, thus failing to distinguish between heavy multimedia applications and other computational tasks.

We suggest a novel approach that solves this problem as follows: Firstly, by monitoring relevant I/O device activity we manage to approximate the "volume of user-interaction" associated with each process. Secondly, by monitoring interprocess communication, we manage to deduce the closure of processes relating to the I/O devices and hence to the user (directly or indirectly). Lastly, we define a scheduler that uses the above information to prioritize tasks in such a way that allows interactive and multimedia processes to achieve good results even under extreme load conditions. We claim that this automatically identifies the user's interests and wishes.

Our work includes a full implementation of this scheduler and measurements confirming that it indeed meets its goals. The scheduler was implemented within the Linux X-Windows environment. The implementation involved some modification of the X-server and a complete rewrite of the Linux scheduler.

1 Introduction

Prevalent commodity systems use a simple scheduling scheme that has not changed much in 30 years. Processes are scheduled in priority order, where priority has two main components: static and dynamic. The static component reflects inherent importance differences (e.g. system processes might have higher initial priority than user processes). The dynamic part depends on CPU usage and ensures that priority of a process is lowered proportionally to the amount of CPU cycles it consumes. CPU usage is forgotten after some time, in order to focus on recent activity instead of distant history.

Tying priority to lack of CPU usage achieves two important goals. The obvious one is fairness: all active processes get a fair share of the CPU. The second one is responsiveness: the priority of a blocked (I/O-bound) process grows with time, so that when it is awakened, it has higher priority than that of other (CPU-bound) processes and is therefore scheduled to run immediately; in fact, in most systems this is the *only* mechanism that provides responsiveness for I/O-bound processes. This was sufficient in the past, when user-computer interaction was mainly conducted through text editors, shell consoles, etc. — all applications that exhibit very low CPU consumption. Nowadays, computer workloads (especially on the desktop) contain a significant multimedia component: playing of music and sound effects, displaying video clips and animation, etc. These workloads are not well supported by conventional operating system schedulers [15], as multimedia applications are very demanding in terms of CPU usage and therefore indistinguishable from traditional background (batch) jobs.

Figure 1 is a good example of this deficiency. It demonstrates what happens when a Xine movie-player displays a short clip along with an increasing number of CPU-bound processes (which we call *stressors*) executing in the background. When no such processes are present, Xine gets all the resources it needs (which is about 40% of the CPU). Adding one stressor process is still tolerable since it takes the place of the idle loop. But after that, each additional stressor reduces Xine's relative CPU share, and causes a significant decline in its displayed frame rate. For example, when 4 stressors are present, each gets about 15% of the CPU, and Xine only gets about 20% (half of what it needs), thereby causing the frame rate to drop by a bit more than 50%.

In recent years, there has been increasing interest in supporting multimedia applications. Several solutions were proposed to the above problem, which fall into two main categories. The first involves specialized APIs that enable applications to request special treatment, particularly in the area of real-time support, and schedulers that respect these requests [16, 7, 18, 10]. The major drawback of such an approach is that it reduces portability and requires larger learning and coding effort. The second category implements support for quality of service in the kernel, and allows users to explicitly control the QoS provided to different applications [6]. While this does not

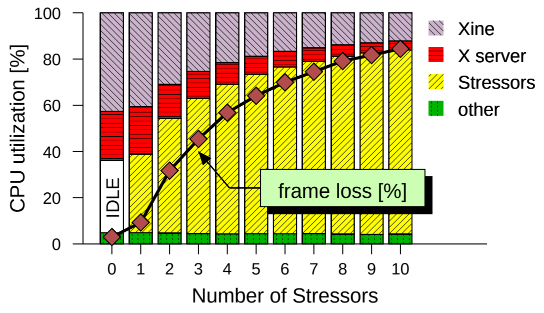

Figure 1. *Competition between the Xine movie player and background stressor processes causes Xine to receive less CPU resources as more stressors are added, resulting in increased frame loss rates.*

require any modifications in the application, it shifts the burden of configuring the system to the user, who must cater for each application individually. This is probably not a good solution for transient interactive and multimedia tasks that come and go during normal work.

Our approach tackles the problem from a different angle. We start by recognizing the fact that some I/O devices can supply us with a fairly good approximation of the user's interests and wishes. We can assume with a reasonable degree of certainty that when the user types on the keyboard, he wants the target application to receive this input and respond to it in a timely manner. Similarly, if some application continuously produces output that spans a significant portion of the screen, it wouldn't be too far fetched to conjecture that the user is interested in this output. By getting such data from the relevant I/O devices, it is possible for the system to identify applications that are of immediate interest to the user, and prioritize them accordingly. In particular, this solves the problem shown in Figure 1; the results, in which Xine is prioritized relative to the stressors and retains its full frame rate are shown in Figure 7.

Importantly, this approach handles both traditional interactive applications (such as text editors) and modern multimedia applications: both types can be identified by tracking user I/O. We collectively denote such applications as being *Human Centered*, or *HuC* for short.

The rest of this paper is organized as follows. Section 2 describes the test platform and the workload we used. Section 3 argues that identification of HuC applications based on CPU consumption patterns fails to deliver. Section 4 describes our alternative identification mechanism based on monitoring relevant I/O devices. Sections 5 and 6 describe the new scheduler that makes use of the new information obtained, and present the results achieved by this scheduler. Section 7 surveys related work and Section 8 discusses our conclusions.

2 Methodology and Applications

Before presenting the HuC scheduling scheme, we first describe the experimental platform and introduce the applications used to evaluate the newly proposed scheduler.

2.1 The Test Platform

Most measurements were done on a 664 MHz Pentium 3 machine equipped with 256 MB RAM and a 3DFX Voodoo3 graphics accelerator with 16 MB RAM that supports OpenGL in hardware. The operating system is a 2.4.8 Linux kernel (RedHat 7.0), with the XFree86 4.1 X server. The clock interrupt rate was increased from the default 100Hz to 1,000Hz. This clock rate has already been adopted in the current Linux kernel development version, and is more suitable for multimedia applications which require millisecond timing resolution [16]. We have also verified that the increase in overhead is negligible.

2.2 The Kernel-Logger Utility

The measurements were conducted using *klogger*, a kernel logger we developed that supports fine-grain events. While the code is integrated into the kernel, its activation at runtime is controlled by applying a special *sysctl* call using the /proc file system. In order to reduce interference and overhead, logged events are stored in a sizeable buffer in memory (typically 4MB), and only exported at large intervals. This export is performed by a daemon that wakes up every five seconds. The implementation is based on inlined code to access the CPU's cycle counter and store the logged data. Each event has a 20-byte header including a serial number and timestamp with cycle resolution, followed by event-specific data. The overhead of each event is only a few hundred cycles leading to a total of about 0.95%. Logging is performed for all scheduling-related events: context switching, recalculation of priorities, forks, execs, changing the state of processes, and monitoring of activity on Unix-domain sockets (to track potential interactions with the X server).

2.3 The Workload

As there are numerous different applications in contemporary desktop workloads, we have identified several dominant application classes and chose to focus on a representative or two from each class.

- **Classic interactive applications**: The (traditional) Emacs and the (newer) OpenOffice text editors. During the test, editors were used for standard typing at a rate of about 8 characters per second.

- **Classic batch applications**: Artificial CPU-bound processes (stressors) and a complete compilation of the Linux kernel. Several processes are involved in compilation and therefore the associated data presented in this paper is a summation. The CPU-bound processes serve as a background load that can absorb any number of available CPU cycles, and compete with HuC processes.
- **Movie players**: MPlayer and the Xine MPEG viewer, which were used to show a short video clip in a loop. While MPlayer is a single threaded application, Xine's implementation is multithreaded, making it a suitable representative of this growing class of applications [8]. Specifically, Xine uses six distinct processes. The two most important ones are the *decoder*, which reads the data stream from the disk and generates frames for display, and the *displayer*, which displays the frames at the appropriate rate. In our experiments, audio output was sent to /dev/null rather than to the sound card, to allow focus on interactions with the X server.
- **Modern interactive applications**: The Quake III Arena role playing game. An interesting feature of Quake is that it is adaptive: it can change its frame rate based on how much CPU time it gets. In our experiments, when running alone it is usually ready to run and can use almost all available CPU time.

In addition, the system runs a host of default processes, mostly various daemons. Of these, the most important with regard to interactive processes is obviously the X server.

3 Identifying HuC Processes Based on CPU Usage Patterns

Traditionally, schedulers on desktop machines prioritized processes based on their CPU usage, or rather, based on lack of CPU usage. The underlying assumption was that I/O-bound processes, which do not use significant CPU resources, may be interactive. In this section we show that such reasoning is obsolete, as modern HuC processes may use significant CPU resources. More generally, we show that it is impossible to distinguish between HuC processes and other processes based on CPU usage patterns alone. To do so, we consider three aspects of CPU usage: the overall CPU consumption, the lengths of effective quanta, and the reasons for relinquishing the CPU.

3.1 CPU Consumption

The simplest measure of CPU usage is total consumption. Most general purpose schedulers base priority

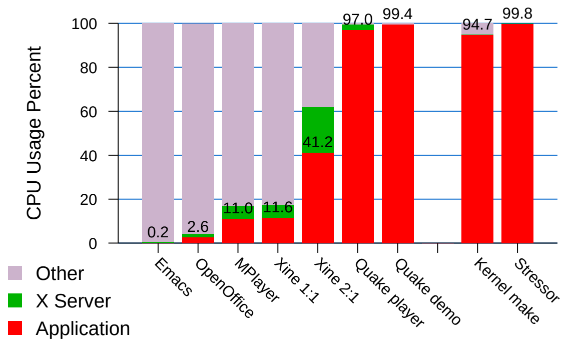

Figure 2. *CPU consumption of different applications expressed as a percentage of the wallclock time. Each application was run alone.*

mainly on this metric. Processes that use the CPU lose priority, while those that wait in the queue gain priority. The details of how this is done is described in Appendix A. for several leading schedulers.

The question, however, is whether low CPU consumption can be used to identify HuC processes. Figure 2 demonstrates that this is not the case. HuC processes are seen to span the full range from very low CPU usage (the Emacs and OpenOffice editors) to very high CPU usage (the Quake role-playing game). Movie players such as Xine provide an especially interesting example: their CPU usage is proportional to the viewing scale. Showing a relatively small movie, taking about 13% of the screen space, required about 15% of the CPU resources for the player and X combined. Using a zoom factor of 2:1, the viewing size quadrupled to about half the screen, and the resource usage also quadrupled to about 60%. Attempting to view the movie on the full screen would overwhelm the CPU. This is despite using an optimization by which the frame data is handed over to X using shared memory.

3.2 Effective Quantum Lengths

While CPU consumption is the main metric used by current schedulers, other metrics are also possible. A promising candidate is the distribution of *effective quantum lengths*. An effective quantum is defined to be the period from when a process is allocated a processor until when the processor is relinquished: either because the process has exhausted its allocated quantum, or because it blocks waiting for some event, or because a newly awakened process has higher priority. The intuition is that although HuC processes may exhibit large CPU consumption, their effective quanta probably remain very small due to their close interaction with I/O devices. Thus we expect to see a difference between the allocated quanta

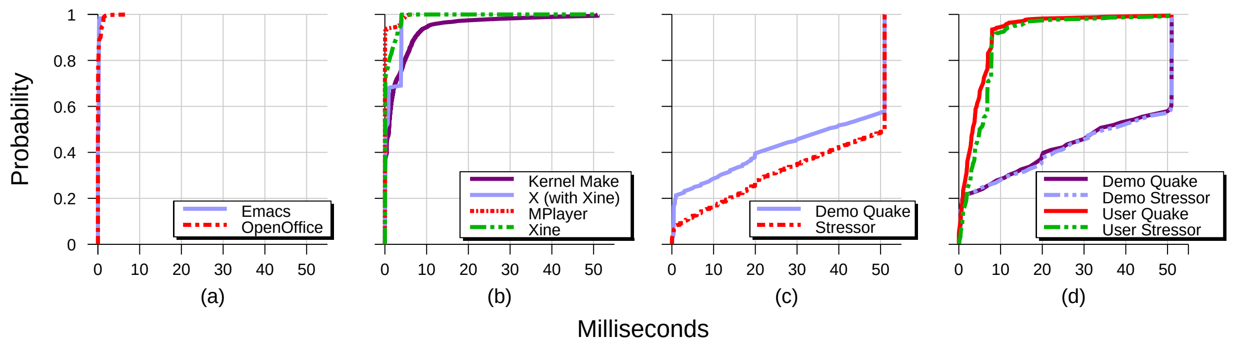

Figure 3. *Cumulative distribution function (CDF) of the effective quanta of the various applications.* **(a)** *Editors have very short effective quanta.* **(b)** *Movie players also have short effective quanta, but this is similar to the profile of the traditional kernel-make batch job.* **(c)** *Quake can consume all available CPU cycles, so when running in demo mode it behaves like a stressor. Both are occasionally interrupted by various system daemons or by the klogger, causing around 50% of the effective quanta to end prematurely.* **(d)** *When a stressor runs with Quake (in demo or user mode), both end up with the same distribution.*

and the effective ones in HuC processes, but expect non-HuC processes to typically use their full allocation.

An example motivating this approach is provided by the computation pattern of movie players. After a movie player decodes the next frame, it sends the X-server a request to display this frame, thus relinquishing the processor until such time when the frame is indeed displayed. Since this scenario repeats itself for each frame, it leads to a ping-pong like computation pattern in which the X-server and the movie player run in an alternating fashion, making the effective quantum orders of magnitude shorter than the allocated quantum. Specifically, our measurements show that most of the effective quanta of such applications are much shorter then 1ms (80% for Xine, and 94% for MPlayer), even though the default quantum length in Linux-2.4 is a bit more than 50ms.

Unfortunately, it turns out that the distribution of effective quantum lengths does not distinguish HuC processes any more than the total CPU consumption does. Figure 3 shows these distributions for different groups of applications. Multimedia applications, in particular, are indistinguishable from other application types: on one hand Quake behaves just like a CPU stressor, both when running alone and when running with a competing process, and on the other hand Xine resembles the well-known kernel-make benchmark. Especially interesting is the case of Quake running together with a stressor. In this case user interactions with Quake (averaging about 165 X-events per second) cause most effective quanta of *both* processes to be shorter than 10ms. This happens because X consumes much less CPU than either of them, and therefore has much higher priority and is able to preempt them whenever it needs to handle an event.

Emacs	Open Office	MPlayer	Xine	Quake user	Quake demo	Kernel make	Stressor
99.6	99.1	98.5	83.1	14.3	1.2	81.6	0.5

Table 1. *Percent of context switches that are voluntary for the various applications.*

3.3 Voluntary vs. Forced Context Switches

Another possible metric is the *type* of context switch. HuC processes (such as movie players) often relinquish the processor voluntarily, due to their dependency on I/O devices, through which they communicate with the user. We can therefore classify processes according to the *fraction* of their effective quanta that ended voluntarily, rather than the *duration* of the effective quanta (as described above).

We define a voluntary context switch as one that was introduced by the process itself, either explicitly by blocking on a device, or implicitly by performing an action that triggered another process to run (such as releasing a semaphore). We were able to trace such context switches by monitoring the various kernel queues. The results shown in Table 1 indicate that this new metric also fails to make a clear distinction between HuC and other processes. Here too, similarities are evident between Quake and stressors on one hand, and between Xine and kernel-make on the other hand.

3.4 Consequences of Not Identifying HuC Processes

To conclude, prioritizing modern HuC processes solely based on their CPU usage no longer delivers, as multimedia and other heavy computational tasks essen-

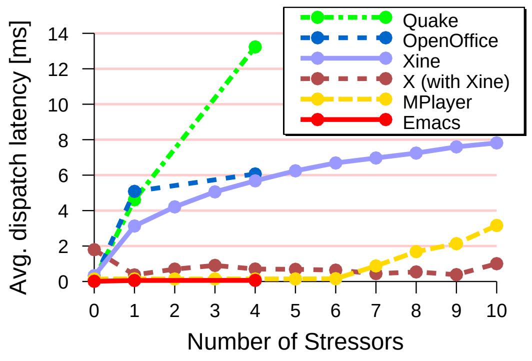

Figure 4. *Average dispatch latency of HuC applications as a function of load.*

tially look the same in that respect. The failure of contemporary general purpose schedulers to differentiate between the two types of processes results in treating them in a similar manner. As the system load increases, the scheduler lacks the ability to favor multimedia processes over the others, consequently depriving them of the resources they need.

In particular, when the system load grows so does the scheduling latency (Figure 4). This is crucial for multimedia applications, as a high scheduling latency may cause them to miss a deadline. In fact, this is one of the reasons for the high frame loss rate shown in Figure 1. To solve this problem we must identify HuC processes directly, and prioritize them accordingly.

4 Identifying HuC Processes Based on User Interaction

Failing to identify HuC processes using the traditional CPU-consumption-based metrics suggests a different approach is needed. It seems there's no alternative to actually follow the flow of information between the user and the various processes, and explicitly characterize HuC processes as such according to the magnitude of this flow. We achieve this using a combination of two mechanisms. The first, described in Section 4.1, is responsible for quantifying the volume of direct interaction between each process and the user. This by itself is insufficient because processes may interact with the user in an indirect manner. This motivates the second mechanism, described in Section 4.2, that tracks interprocess communication to unearth dependency relationships between them. Together, the two mechanisms allow the scheduler to correctly identify and prioritize HuC processes. Similar ideas have been suggested by Flautner et al. [8], but not in the context of scheduling.

4.1 Monitoring User I/O

The concept of a human user is unknown to the kernel. User interaction is mediated by peripheral devices. Identification of user interaction must therefore start with the identification of devices that represent the user.

4.1.1 HuC Devices

Only a subset of the peripheral devices in the system are of interest when trying to quantify the volume of interactions between the user and the various processes. These include the keyboard, mouse, screen, joystick, sound card, etc., and will be referred to collectively as *HuC devices*. The common property of such devices is that they all directly interact with the user. For the purpose of this research we've decided to only monitor the "bare necessities", namely the keyboard, mouse, and screen. The same principles can be applied to the other HuC devices in a straightforward manner.

Unix environments use the X-Windows system [27] as the conventional mechanism to multiplex I/O between the user (as reflected by HuC devices) and the various applications. Applications that wish to communicate with HuC devices are referred to as *X-clients*. Clients connect to the *X-server* and communicate with it using the *X-protocol*. The server usually associates a window (called a *virtual terminal*) with each client, such that user input events performed within this window are forwarded to the client (in the form of *X-events*), and output produced by the client (in the form of *X-requests*) is directed to this window. Consequently, the X server centralizes all work concerning the kernel mechanisms that allow communication with the canonical HuC devices, and hence with the user. It is therefore natural to use the X server as a meta-device when monitoring user I/O.

X maintains in its internal data structures a *client record* for each client. We have added three fields to this record:

1. client's process ID,
2. client's temporal input ratings (denoted C_{tin}), and
3. client's temporal output ratings (denoted C_{tout}),

which are discussed below. The values of these fields are communicated to the kernel once a second (using the X native timer mechanism) through non-standard parameters we've added to the standard POSIX *sched_setparam* system call [9].

Client process IDs (denoted *pids*) are of course needed because eventually the scheduler will base its decisions upon the I/O ratings associated with each pid. X doesn't originally maintain pids, because one of its major design goals is to serve local or remote clients in the same way, and holding the pid of a remote client has no meaning. In the context of desktop scheduling, however, we are only interested in monitoring local clients, since these are the candidates for being HuC processes (the option of running HuC applications remotely in a distributed environment is beyond the scope of the current research effort). To obtain the pids of connecting clients we slightly modified the communication layer of the X server. This is based on the fact that local clients connect to the server via a Unix-domain socket [23], and non-standard Unix-domain socket options implemented in Linux [12] provide access to the sender's pid.

4.1.2 Quantifying User Input

Input events can be perceived as an immediate and explicit expression of the user's wishes. The number of events is typically not so important: dragging with the mouse, which generates multiple events per second, conveys the same amount of user interest as a single mouse button click or the typing of a single character. The most important metric is recency: the most recent user input should get the highest priority.

Reflecting these considerations, we implement input ratings as follows. Whenever an input event is associated with a certain client, that client's C_{tin} counter is initialized to a predefined constant K. Each second, after C_{tin} is communicated to the kernel, it is decremented by 1. Thus a process that receives input retains its HuC status for K seconds, and processes with more recent input have higher priority. The code for this is also simple. X already has a list of callbacks to invoke whenever an input event is read from the device files; we have added another callback that logs this event by setting C_{tin}.

In addition to the regular periodic updates sent from X to the kernel once a second, whenever a client with zero input ratings receives an input event, the kernel is immediately notified. This allows the scheduler to maximize responsiveness by promptly handling such events.

4.1.3 Quantifying Output to the User

Unlike user input that has almost an unary nature (a human user can deliver simultaneous events to very few processes in one second) and which reflects the immediate wishes of the user, quantifying output is a bit more complex: firstly, because various applications may simultaneously produce output to different windows, but more importantly, because we don't know which of these output events is more significant to the user. To cope with this difficulty we exploit a feature in human perception that is a remnant of our predatory days: human vision is more sensitive to movement [19]. By quantifying the rate of changes produced by each client we get a reasonable guess about which process has the user's attention. We further assume that the user does not like to be distracted and will eliminate any source of interference (e.g. iconify an irrelevant window). Anything that grabs the user's attention is assumed to be "important". It is interesting to note that this approach allows users to implicitly control resource allocation by iconifying windows, which corresponds to natural desktop working habits.

The question remains of how to quantify the rate of screen changes. Simple event counting will not work in this case since an output event can be as small as printing a character or as large as changing the background image. To complicate matters even more, output events may refer to hidden portions of windows. Of course changes that can't be seen by the user, shouldn't be included in the client's ratings. Our goal is therefore to approximate the percentage of the screen actually changed due to each output event, and to accumulate this percentage in the C_{tout} of the event's initiator. This is a feasible task since the X protocol defines a reduced set of only seventeen graphical X-requests that are available to clients. Requests include drawing a polygonal line, a character string, an image, etc. For each X-request we have implemented a function that approximates the amount of change it introduces to the screen (for example, when drawing a character, we use its bounding box size as the approximation). Additionally, we've hooked to the X clipping mechanism in order to find out how much of the change is indeed visible to the user. Finally, the resulting change is expressed as a percentage of the screen area and accumulated in C_{tout}.

We remark that even though the X protocol is the conventional paradigm used to perform user I/O in Unix environments, other mechanisms do exist. The *Direct rendering Infrastructure* (DRI [17]) is the dominant alternative since it is used by the *OpenGL* graphical library [20], which in turn is heavily used by graphical software (such as Quake). DRI interacts directly with the graphics controller, thus circumventing the X protocol. This is why we don't include demo-Quake in our measurements below (user-Quake however is still included, since Quake only uses OpenGL for output, and relies on X for input). In order to make our implementation complete, OpenGL should have also been modified (similarly to the X server) to maintain the per-process I/O statistics and to periodically report them to the kernel.

An alternative design would be to maintain the required statistics in the relevant I/O device drivers in kernel space. This approach will eliminate the need to change OpenGL or other similar libraries. Note however that this approach is very hard to implement because device drivers are not aware of windows structure and of high level operations. Additionally, this approach will not eliminate the need to change X (as described above), because without this change, all I/O activity proxied through X will be attributed to X itself instead of to its clients. Finally, another serious drawback of this approach is that there are a lot more display adaptor drivers that will need to be modified than there are graphic libraries similar to OpenGL.

4.2 Process Dependency Graphs

As noted above, processes may interact with the user in an indirect manner. In a Unix system, the main process that interacts with the user is the X server. HuC applications interact with the user indirectly, using X as an intermediary. The second component of identifying HuC processes is therefore finding the transitive closure of the processes that enjoy direct interaction. To do so, we must first identify the graph of process interactions.

4.2.1 Identifying Process Interactions

Process interactions may take different forms: communication using a pipe, storing to and loading from shared memory, the use of semaphores, etc. While all these mechanisms are in some way mediated by the kernel, keeping track of all of them is very arduous. Moreover, if new mechanisms are introduced, they will require separate monitoring.

The alternative is to use a single mechanism that allows the kernel to *deduce* that an interaction has taken place. We chose to monitor insertions into the ready queue for this purpose. When one process causes another to enter the ready-to-run queue, it implies that the second process was waiting for the first one, and hence that they interact with each other.

Implementing this idea in Linux is very simple, because attempts to insert a waiting task to the ready-to-run queue are always performed via the *try_to_wake_up(process)* function. This function is usually invoked on processes populating the waiting queue associated with the awaited event. Additionally, this function may sometimes be invoked for processes that are already in the ready-to-run queue (which explains the "try" component in its name), e.g. when a process signals another process and the latter isn't blocked-waiting for that signal (but rather engaged in some other activity). In any case, an invocation of *try_to_wake_up* signifies a dependency between the process that triggered the call and the targeted process. A similar idea is currently being explored by leading Linux developers [24].

However, this heuristic has the obvious drawback that some dependencies might go unnoticed. This happens whenever the communicating processes don't block on the communication. Examples include explicit non-blocking communication such as using shared memory, and situations in which blocking is not necessary (e.g. reading less data than what is available in a pipe). We partly deal with the first case, shared memory, by considering a set of processes that share their address space as a single entity, rather than considering each of them individually.

Even with the above flaws, the approach we've taken manages to produce excellent results in deducing inter-process relations. Its success may be attributed to its self correcting nature: When the system is under-loaded, it doesn't really matter whether our scheduler correctly identifies dependencies or not, as processes get the CPU time they require anyway. However, when the load increases, HuC processes that are not identified correctly begin to suffer (namely spend more time in various waiting queues), thus allowing our scheduler to witness and take into account the circumstances in which they are reinserted into the ready-to-run queue.

4.2.2 The PDG_{in} and PDG_{out} Graphs

X reports to the kernel about processes it directly interacts with, thus allowing the scheduler to identify them as HuC. However, this is not enough, since these processes may depend on other processes that are totally unknown to the X server. Figure 5 demonstrates why this is commonly the case in Unix environments. It describes a scenario in which the user writes some document using the *VI* text editor from within an *xterm* console emulator. When the user presses a keyboard key, the X server reads the associated character from the keyboard's device driver, sends it as an X-event message to *xterm*, which in turn forwards it through a pseudo-terminal connection [22] to *VI*. The latter performs the necessary processing and may update the user's view by propagating data in the opposite direction. This simple example highlights the fact that the HuC quality has a transitive nature, and therefore its definition must be refined such that it will also include processes that indirectly interact with the user. We therefore define a process as being HuC if it directly interacts with the user (through HuC devices) or if it interacts with other HuC processes.

As described above, we identify process interactions by noting which processes attempt to insert other processes into the ready queue. This induces a directed graph on the processes. We call this the *Process Dependency Graph* $PDG = \langle V, E \rangle$, where V contains all active processes in the system, and E contains edge (P_i, P_j) iff P_j was recently inserted to the ready-to-run queue due to an action taken by P_i.

PDG as defined includes the raw dependency data. But we need to distinguish between dependencies that reflect the flow of input from the user to a process, and dependencies that reflect flow of output from a process to the user. To do so we define two other graphs, PDG_{in}

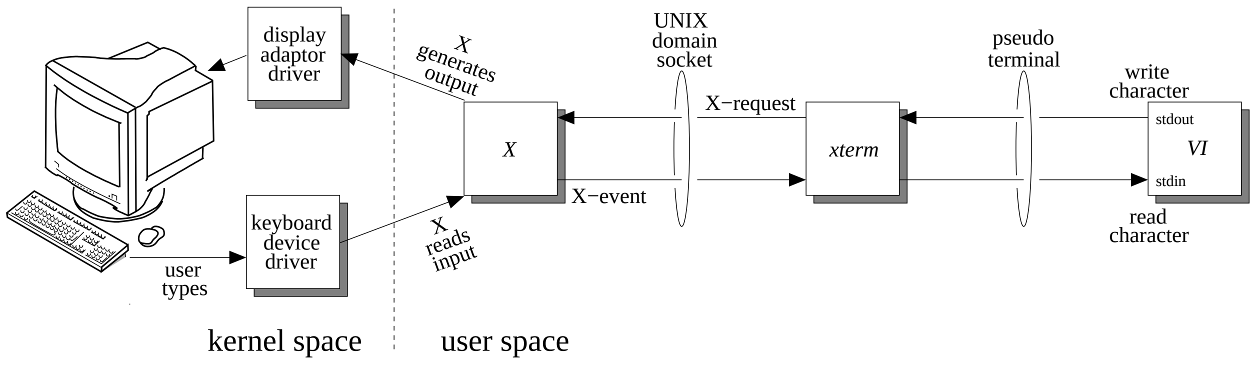

Figure 5. *Information flow between the user and VI.*

and PDG_{out}. In each of these graphs we seek the connected component that includes the X server. The connected component of X in PDG_{in} will identify all HuC processes that receive some user input, and the connected component of X in PDG_{out} will identify HuC processes that generate output to the user.

PDG_{in} is actually the same as the original raw PDG. The reason is that input flows form the X server to other processes, and exposes process dependencies along the way. To demonstrate this, let us reexamine the example given in Figure 5. After reading input from the keyboard device driver, X generates a suitable X-event and writes it through a socket to the waiting (blocking read) *xterm*, consequently removing it from an internal operating system waiting-queue and inserting it to the ready-to-run queue: its socket end is now ready for reading. This means that $(P_{Xserver}, P_{xterm}) \in PDG$. Likewise, xterm sends the keystroke to VI, which is blocked waiting for it, and causes it to unblock, so $(P_{xterm}, P_{VI}) \in PDG$. We get that the connected component of X in PDG (and hence in PDG_{in}) is indeed correctly composed of $\{P_{Xserver}, P_{xterm}, P_{VI}\}$.

This reasoning does not work for output. Consider output flowing from VI to the screen. As VI generates output and sends it to xterm, we get $(P_{VI}, P_{xterm}) \in PDG$. Xterm then sends a request to X, so we get $(P_{xterm}, P_{Xserver}) \in PDG$. But if we try to find the connected component of X in this case we find that it only contains X itself, because the arcs are oriented in the wrong direction:

$$Xserver \leftarrow xterm \leftarrow VI$$

We therefore define PDG_{out} to be the *inverse* of the raw PDG: $(P_i, P_j) \in E_{out}$ iff $(P_j, P_i) \in E$. It contains the same arcs, but in the opposite direction.

4.2.3 Aging the Data

Interactive processes may accept input at one time, generate output at another, and just compute in between. If the computation is long, it is not justified to retain the HuC status indefinitely. Thus the arcs in the PDGs should only reflect "recent" interactions.

In order to actually maintain the graphs, we need to define the meaning of "recent". For this purpose we have defined a non-negative integral weight function on the graph's edges, such that the weight of edge (P_i, P_j) is the number of times P_j was inserted to the ready-to-run queue due to P_i (each such event increments the associated edge's weight by one). To make weights reflect recent history only, they are exponentially decayed by a factor of two, once a second. The actual definition of the PDGs is therefore:

$$PDG_{in} = \{(P_i, P_j) \mid weight(P_i, P_j) > 0\}$$

$$PDG_{out} = \{(P_i, P_j) \mid weight(P_j, P_i) > 0\}$$

When considering our *VI* example, this means that if the user has typed eight characters per second and then stopped, the edges generated due to this input burst will sustain a positive weight for three seconds, after which the scheduler will cease recognizing *VI* as HuC. Arguably this is a very short period of time. However, this is not a problem since according to our measurements, processes that produce a lot of output (such as movie players) continuously sustain heavy edges in the connected component of X within PDG_{out}. As for the more traditional HuC processes (such as *VI*), recall that X immediately notifies the kernel whenever a client with zero input ratings receives an input event (Section 4.1). This allows the kernel to immediately adjust priorities according to the consequently generated edges, instead of relaying on history (as will shortly be explained).

Of course using less aggressive decaying policies may also be considered. One reasonable alternative is a linear decay of weights: decrementing weights by one each second instead of dividing them by two (such that decaying a burst of eight typed characters will take eight seconds). Another reasonable alternative is to break the connection between the number of times P_i inserted P_j to the ready-to-run queue and the duration the associated edge maintains positive weight. Instead, we can decide that a dependency (an edge) is preserved for some fixed predefined duration T, and change the definition of weight to be a second countdown (initialized to T) from when the most recent dependency was recorded.

4.3 Propagating I/O Ratings

Recall that X sends the kernel information regarding the I/O ratings of different processes once a second. But this relates only to processes that interact directly with X. We therefore need to propagate the ratings to all the other processes in X's connected components in the PDGs. The following describes how propagation is actually done.

The Linux kernel represents each task using a structure to which we have added four fields: input rating (denoted P_{in} for process P), temporal input rating (P_{tin}), output rating (P_{out}), and temporal output rating (P_{tout}). All these fields are initialized to zero at boot time. After the kernel receives an update from X, it uses the temporal input component to compute the associated ratings as follows:

1. Let Ω_{in} denote the set of processes that X identified as heaving positive input rating ($= C_{tin}$ defined in Section 4.1). We start by initializing P_{tin} with the associated C_{tin} for each process $P \in \Omega_{in}$.
2. Next, by applying a breadth-first search (BFS) from each $P \in \Omega_{in}$, we propagate P_{tin} through PDG_{in} in a cumulative manner. This means that if a process P^1 is reachable from two processes $P^2 \in \Omega_{in}$ and $P^3 \in \Omega_{in}$, then $P^1_{tin} = P^2_{tin} + P^3_{tin}$ at the end of this phase. P^1 has higher rating than P^2 and P^3, because both of them depend on P^1.
3. Finally, by traversing through *all* processes with positive input ratings (temporal or actual), we update: $P_{in} = \frac{P_{in}+P_{tin}}{2}$.

The overhead for running this is actually quite low, as the PDG is typically sparse and the connected component is small.

Figure 6 presents an example of the above algorithm. The left graph presents X's connected component in PDG_{in}, with uppercase letters denoting input ratings (P_{in}) before the propagation algorithm was executed. Assume X has just reported to the kernel that the temporal input ratings of P^D, P^E, and P^F are d, e and f, respectively (and therefore $\Omega_{in} = \{P^D, P^E, P^F\}$). Lowercase letters in the right graph denote temporal input ratings (P_{tin}) after propagation. For example $P^J_{tin} \equiv j = d + e + f$ because P^J is reachable from all processes in Ω_{in}. The content of nodes in the right graph is of course their new input ratings.

Output ratings are computed similarly (and are normalized to the same scale as the input ratings). Non-temporal I/O ratings are inherited upon *fork*.

Note that it is very common for PDGs to include cycles, as processes that consume input usually also produce output (see Figure 5). Additionally, all nodes of Ω_{in} are almost always reachable from X within the PDGs. Our breadth-first search is therefore prohibited from going through X's node, or else every node's temporal rating will contribute to every other node's rating, resulting in equal ratings across the entire component.

In addition to the periodic executions of the above propagation algorithm, it is also invoked whenever X reports on a process with zero input rating that has just consumed an input event (recall that such processes are reported immediately to the kernel). When this is the case, the breadth-first search is only conducted from the newly reported process, rather then from all processes in Ω_{in}. To complement this special treatment, *try_to_wake_up* (through which all insertions to the ready queue are done) is modified accordingly: if the inserting process is HuC and the inserted one is not, the I/O ratings of the former are copied to the latter. Backtracking to the example given in Figure 5, this mechanism ensures that upon input, *VI* will immediately be identified as HuC, even if it currently has zero I/O ratings. The immediate feedback of X about *xterm* is instantly propagated to *VI* when the former inserts the latter to the ready queue.

5 Linux Implementation of the HuC Scheduler

Our implementation has two components. The first is modifications to the X server, which monitor direct user I/O with various processes. This is communicated to the kernel once a second using a well-defined interface. The second component is a modified scheduler, that creates the PDGs and prioritizes processes accordingly.

5.1 Scheduling-Classes Hierarchy

The Linux scheduler is POSIX compliant and therefore supports three scheduling classes: FIFO, Round-Robin, and OTHER (the latter is not defined by POSIX but its implementation is mandated and it is the default [9]). Each process is associated with a single class that can be changed through the standard *sched_setparam* system call. FIFO and Round-Robin processes are categorized by POSIX as realtime and when ready to run, should always be preferred over OTHER processes. Unfortunately, in Linux all three schedulers are hard-coded into one complex function which makes it very tricky to add adequate handling for HuC processes. For this reason we have decided to rewrite the scheduler in such a

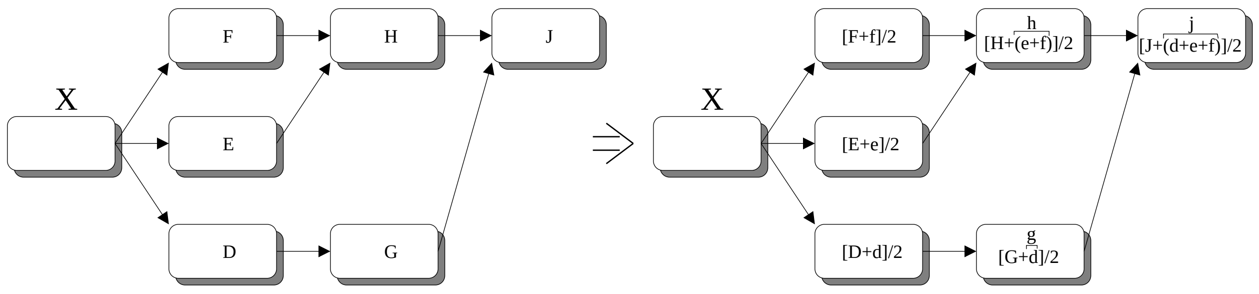

Figure 6. *Example of the propagation of input ratings.*

Class	Description
FIFO	POSIX First-In First-Out
RR	POSIX Round-Robin
KTHREAD	kernel threads
HUCIN	recently consumed user input
HUCOUT	recently produced output viewed by the user
OTHER	Linux default

Table 2. *Scheduling classes hierarchy ordered by importance.*

way that will allow new policies to be easily incorporated into the kernel. Our simple design was inspired by that of the Solaris 8 scheduling scheme [13] and can be described as a *hierarchical scheduler*: The various scheduling classes are organized in a hierarchy, in order of importance. Whenever the scheduler needs to choose the next process to run, it traverses the hierarchy in order, “asking” each class to pick its most desirable process. If no process is eligible to run in the current class, the scheduler moves to the next class, until an appropriate process is found.

Table 2 lists the scheduling classes we have implemented in our scheduler. We have added the new KTHREAD, HUCIN, and HUCOUT to the existing FIFO, RR, and OTHER. KTHREAD is populated by the various kernel processes which may be considered as part of the operating system (swapper etc.). Depriving such processes of a processor might sometimes have a disastrous effect on the system. Prioritizing within the KTHREAD class is done identically as in the original OTHER class. HUCIN includes all processes with positive input ratings. HUCOUT includes all processes that have positive output ratings (and are not already included in HUCIN). By placing HUCIN above HUCOUT we acknowledge the fact that input events reflect the immediate and explicit expression of what the user wants. Prioritizing within the two HuC classes is very similar to the default, with the difference that when a new epoch is started (see Appendix A.) durations of newly allocated quanta are set to be related to the associated rating. The process with the highest output rating in HUCOUT will be allocated a full default quantum duration (≈ 50ms). Processes with lower rating will get a lower allocation, but more than their proportion (e.g. a rating of 0.3 of the maximal rating can lead to an allocation of 0.5 of the maximal allocation). This allows new processes to build up their output rating.

5.2 Handling Positive Feedback

Traditional Unix schedulers are stable because they include a negative feedback loop. High priority processes get to run and lose priority, whereas waiting processes gain priority. As a result active processes quickly converge to the same priority level and share the CPU equitably.

Our scheduler has an inherent unstable positive feedback loop: processes that consume input and generate output get a higher priority, which allows them to run more, potentially creating even more output (input depends on the user). Thus a new HuC process may be unable to get started and gain enough momentum to compete with existing HuC processes. However, this seems not to be a problem in practice, because new processes inherit the user-interaction counts of their parents. As new processes are typically created as a result of a user request, the process which creates them (shell or GUI) becomes a HuC process, and the new process will start as a HuC process.

Nevertheless, a more general solution is desirable, e.g. to handle processes forked by non-HuC processes or processes that change their nature over time. Such a solution must allocate CPU time to processes despite the fact that their I/O ratings are low or nil. Luckily, this meshes in nicely with the Linux concept of an epoch. During an epoch, all active processes get to run. Rather than defining the epochs within each scheduling class, we can define an epoch to span all classes. The allocation of time within the epoch, however, depends on the class: HuC processes will get a much higher allocation than OTHER processes. The ratio between the allocations is a configurable system parameter. While essentially straightforward, this idea has not been implemented yet.

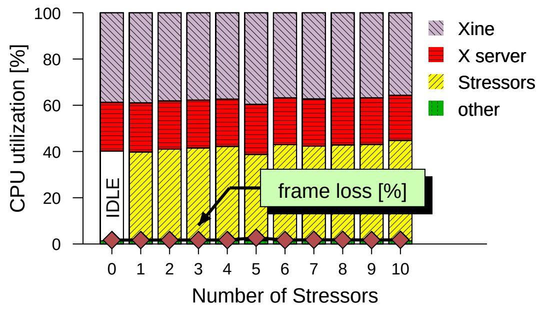

Figure 7. *Division of CPU time and Xine's frame-loss rate under the HuC-scheduler (compare with Figure 1).*

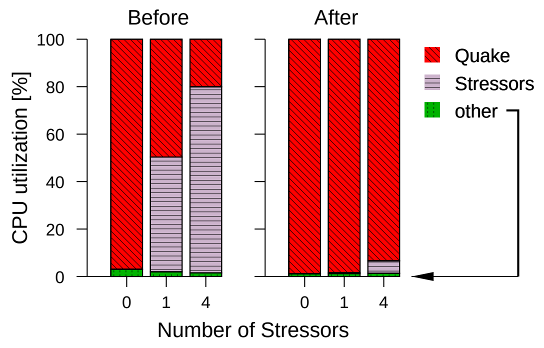

Figure 8. *CPU share given to quake by the default scheduler (left) and by the HuC scheduler (right).*

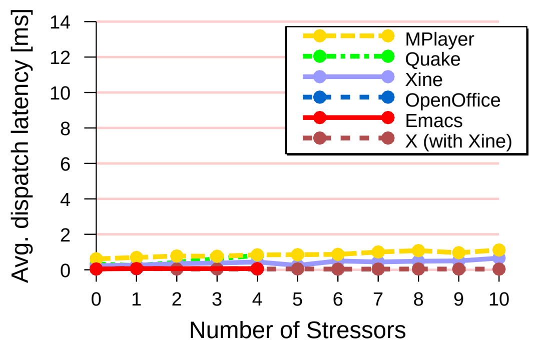

Figure 9. *Average dispatch latency of HuC applications under the HuC-scheduler (compare with Figure 4).*

6 Experimental Results

To evaluate the concept of HuC scheduling and our Linux implementation of this concept we conducted measurements with several workloads. The workloads typically included at least one HuC process, and different numbers of stressor processes that compete for the CPU.

Probably the most striking result is shown in Figure 7. This shows profiles of executing Xine showing a movie at a 2:1 size ratio, with up to 10 stressor processes. Xine and the X server require about 60% of the CPU in this case. Under the original Linux scheduler, they do not get this percentage when there are two or more stressors, resulting in an increasing frame-loss rate as stressors are added (Figure 1). But with the HuC scheduler Xine and X are identified and given priority over the stressor processes, and they continue to get 60% of the CPU regardless of the number of stressors. As a result the frame loss rate remains negligible.

Similar results are obtained for other applications as well. At the low end of CPU usage, applications like the Emacs editor are unaffected by the HuC scheduler. Emacs only requires maybe 1% of the CPU resources, and gets it even under the default scheduler; the HuC scheduler provides the same. But at the high end, Quake can adaptively use CPU resources to improve its output quality. When run under the default scheduler, its share of the CPU is reduced when stressor processes are added. With the HuC scheduler, it can continue to dominate the CPU (Figure 8).

The HuC scheduler not only allocates CPU time preferentially to HuC processes, it also does so promptly. Recall Figure 4, which showed the dispatch latency of various process types under loaded conditions when served by the Linux scheduler. Figure 9 shows the results of running the same experiment with the HuC scheduler. The dispatch latencies of HuC-processes remains very low ($\leq$ 1ms), regardless of the background load.

Another point worth mentioning in this context is the improved responsiveness of the window-manger itself. While conducting measurements involving heavy background load under the default scheduler, we have noticed that moving windows around produces extremely jerky and abrupt results. By contrast, the HuC scheduler impressively rectified this misfeature: identifying the window-manager as HuC allowed smooth window movement which (subjectively) felt as if no background load was present.

Finally, we ran a measurement of overloading the machine with HuC processes to check the division of resources between them. Running two Xine players at a 2:1 ratio requires about 120% of the CPU for a rate of 25 frames/sec each (from Figure 2). Figure 10 shows that indeed together they are limited to about 40 frames/sec. We can also see that within a few seconds they settle into a pattern in which one gets slightly more CPU than the other. This corresponds to the fact that their windows partially overlap, so the one on the bottom is partly ocluded and gets a lower priority.

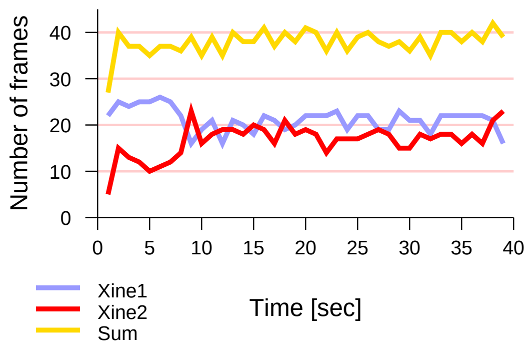

Figure 10. *Frame rates achieved by two competing Xines.*

7 Related Work

Commodity operating systems are only beginning to recognize the importance of user interaction explicitly. For example, Solaris and Windows give processes that wake up after waiting on a device a priority boost that is inversely proportional to the device speed. More advanced support for special situations exists only in research prototypes, and falls into several categories.

The simplest approach is to prioritize the X server [16], or modify it to prioritize clients as the kernel does [6]. However, this by itself does not solve the problem.

The alternative is to modify the kernel scheduler. The first category here is to provide soft real-time support so that multimedia applications can sustain frame rates and audio sample rates. This has two aspects: high resolution timing services, and appropriate scheduling; we focus on the scheduling here. The SMART scheduler [16] by Nieh and Lam lets a multimedia application request the operating system for certain soft real-time assurances — mainly computation periods — and receive feedback from the operating system whether these requests can be met. This has the drawback of requiring application modifications. This problem was addressed by the BEST scheduler [2], which tries to identify applications with periodic computation needs automatically. On a different note, Zhang et al. [28] attempt to schedule real-time jobs along with best-effort ones in a manner which will maintain the real-time deadlines. Their proposed solution is dividing the CPU time among the two classes according to a user supplied "fairness" ratio, and letting each class schedule its processes in a hierarchical model. The real-time class uses the earliest-deadline-first (EDF) scheme.

The second category is explicit support for proportional or fair-share scheduling [6]. This approach shifts the burden to the user, and requires a specification of the resource requirements of select applications. One of the well known schemes in this field is *Lottery Scheduling* [26]. The basic idea is to assign each process a number of lottery tickets that is proportional to its requested percentage of the CPU time. The scheduling decision is than made by drawing a uniformly distributed ticket thus giving each process a chance to "win" the CPU that is distributed according to the user/programmer's requests. Another well known work in this field is the *Borrowed-Virtual-Time* scheduler [7]. This scheduler assigns a virtual time to each running thread, and allocates the CPU according to a user-defined weight policy. A time sensitive thread can "borrow" time from its future allocations when its schedule is tight. The Eclipse operating system [4] went a step further in providing proportional share of the machine, by supporting guaranteed portions of multiple resources at once: not only the CPU, but also memory blocks, disk bandwidth, and network bandwidth. In particular, the algorithm is designed so that latencies incurred waiting for different resources do not accumulate over time.

A third approach is to use hierarchical schedulers — schedulers that allocate CPU time between other, class specific schedulers. Goyal et al. [11] designed a hierarchical scheduler fashioned like a tree. Each node symbolizes a meta-scheduler with specific proportions of CPU time division between its children. The *Vassal* project [5] from Microsoft includes a hierarchical scheduler with strict ordering: when a dispatch decision is to be made the dispatcher queries the various class-specific schedulers for processes, in a strict, predefined order. The focus of this work is to enable the dynamic loading and unloading of schedulers at runtime, according to the applications' demands. This principle is somewhat similar to the hierarchical scheduler we described in Section 5.

Although some of these schemes are very elaborate, there is one common downside to all of them: they still require the user/programmer to manually specify the needs of the various application — either in terms of deadlines or of relative weights — and use this hand tuned process identification as a guide by which to distribute resources. The only exception to this approach is the *BEST* scheduler [2] that tries to automate the identification of soft real-time processes and their requirements, rather then letting the user manually supply this information. But even this provides an automation that is limited to soft real-time processes, cannot distinguish between interactive and non-interactive processes, and cannot handle overloaded machines. Our work, by contradistinction, is the first to fully automate the identification of HuC processes, and make the computer anticipate the user, his desires and interests. Interestingly, the idea of identifying HuC processes as the closure of processes that interact with the X server has been proposed by Flautner et al. [8] — but was not used in the context of operating system scheduling.

8 Discussion and Conclusions

To summarize, the main observation leading to our work is that it is impossible to use CPU usage patterns to identify processes that are of immediate interest to the user. Instead, it is better to directly track the activities of the user, and compute the closure of processes that participate in user interactions. These processes, which we call human-centered, are then prioritized relative to other processes in the system.

Naturally, the implementation of this idea involves many details. We have argued that input from the user should take precedence over output to the user, both because it more directly reflects user interest and because it is easier to quantify. We have developed means to quantify the impact of output, and to propagate I/O ratings among interacting processes. But for each of these mechanisms we also noted that alternative options are possible. The detailed study and comparison of these options are left for future work. In addition, we intend to check the possibility of (and benefits from) better tracking of process interactions, e.g. to include interactions via shared memory. Finally, there is a need for serious study of cognitive aspects of user interaction, e.g. regarding the assignment of different levels of importance to different interaction modes.

The experimental results are very promising, and show that we can indeed identify HuC processes and prioritize them relative to other processes. But this also raises the question of being too successful. Consider a situation in which a user is waiting for the output of a long computation, and engages in an interactive game to pass the time. Our scheduler will identify the game as more important, and might therefore deprive the awaited computation. On the other hand, being based on I/O events also allows our scheduler to respond to very simple cues from the user. Simply clicking on a window will cause an X-event to be sent to the associated application, and will raise its priority, even if it completely ignores the actual input. This opens intriguing possibilities for new types of interactions between the user and the system.

The human-centered nature of the HuC scheduler allows the system to focus on the interactive user's immediate interests. This facilitates the prioritization of interactive and multimedia applications relative to background tasks. Importantly, this includes not only background tasks generated by the same user, but also various system daemons and even remote work. Thus using a HuC scheduler can help in the implementation of grid environments, in which spare CPU cycles not used by the local user are made available to remote work, with the assurence that such remote work will not interfere with the desktop interactivity.

A. Review of Scheduling Algorithms

This review demonstrates how CPU usage is factored into the scheduling algorithms of contemporary operating systems.

The simplest example is the **Traditional Unix** scheduler [1]. The scheduler chooses processes based on priority, which is calculated as the sum of three terms: a *base* value that distinguishes between user and kernel priorities, a *nice* value (partially configurable by the ordinary user to reflect relative importance), and a *usage* value. Lower numerical values represent higher priorities. The usage is incremented on each operating system clock tick for the currently running process, so priority is reduced linearly when a process is running. On the other hand the accumulated usage of all processes is divided by a factor once a second, thus raising their priorities. The factor depends on the load: when load is high, and the process gets to run less often, the aging is also slower. **BSD Unix**, which is the basis for FreeBSD and Mac OS-X, uses a similar formula [14].

In **Linux** the priority dictates both which process is chosen to run, and how long it may run [3, 25].The Linux scheduler partitions time into epochs. In each epoch, every process has an allocation of how long it may run, as measured in ticks. When the process runs, the allocation is decremented on each tick until it reaches zero. Then, the process is preempted in favor of the ready process with the highest positive allocation. When there are no ready processes with an allocation left, a new epoch is started, with all processes getting a new allocation that is inversely proportional to their nice value (the lower the nice value, the higher the priority and thus the higher the allocation). In addition, processes that did not use up all their previous allocation transfer half of it to the new epoch. Thus processes that were blocked for I/O get a higher total allocation, and hence a higher priority.

Solaris is somewhat more sophisticated [13].The Solaris scheduler supports scheduler modules, so new modules can be loaded at runtime by the administrator, thus changing the behavior of the scheduler. The default classes are time sharing (TS), interactive (IA, which is very similar to TS), system (SYS), and real-time (RT). User threads are usually handled by the TS and IA classes. Priorities and quanta are set according to a scheduling-class-specific table, which sets *(i)* the quantum length for each priority, *(ii)* the priority the thread will have if it finishes its quantum (lower), or *(iii)* if it blocks on I/O (higher). The quanta are in operating system clock tick units, and the values in the tables can be changed by the administrator. The basic idea is that higher priorities get shorter quanta: when a process finishes its quantum it gets a longer one at lower priority, and when it blocks it receives a shorter quantum at a higher priority, as opposed to what might happen under Linux.

The priority of threads in **Windows NT4.0/2000** also has static and dynamic components [21]. The static component depends on the thread's type. The dynamic component is calculated according to a set of rules, that may also give the thread a longer quantum. These rules include the following:

- Threads associated with the focus window get a quantum that is up to three times longer than they would otherwise.
- Threads that seem to be starved get a double quantum at the top possible priority, and then revert to their previous state.
- Threads that wait for user input get a double quantum at a priority level that is one less than the maximum, and then revert to their previous state.
- After waiting for I/O, a thread's priority is boosted by a factor that is inversely proportional to the speed of the I/O device. This is then decremented by one at the end of each quantum, until the original priority is reached again.
- Users may specify the relative importance of applications.

References

[1] M. J. Bach, *The Design of the UNIX Operating System*. Prentice-Hall, 1986.

[2] S. A. Banachowski and S. A. Brandt, "*The BEST Scheduler for Integrated Processing of Best-Effort and Soft Real-Time Processes*". In *Multimedia Computing and Networking*, January 2002.

[3] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly, 2001.

[4] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz, "*The eclipse operating system: providing quality of service via reservation domains*". In *USENIX Technical Conf.*, pp. 235–246, 1998.

[5] G. Candea and M. B. Jones, "*Vassal: loadable scheduler support for multi-policy scheduling*". In *2nd USENIX Windows NT Symp.*, pp. 157–166, USENIX, Seattle, WA, August 1998.

[6] S. Childs and D. Ingram, "*The Linux-SRT integrated multimedia operating system: bringing QoS to the desktop*". In 7th *Real-Time Technology & App. Symp.*, p. 135, May 2001.

[7] K. J. Duda and D. R. Cheriton, "*Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler*". In 17th *Symp. Operating Systems Principles*, pp. 261–276, Dec 1999.

[8] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge, "*Thread-level parallelism and interactive performance of desktop applications*". In 9th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 129–138, Nov 2000.

[9] B. O. Gallmeister, *Posix. 4: Programming for the Real World*. O'Reilly & Associates, January 1995.

[10] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole, "*Supporting time-sensitive applications on a commodity OS*". In 5th *Symp. Operating Systems Design & Implementation*, pp. 165–180, Dec 2002.

[11] P. Goyal, X. Guo, and H. M. Vin, "*A Hierarchical CPU Scheduler for Multimedia Operating Systems*". In *Usenix 2nd Symp. on Operating Systems Design & Implementation (OSDI)*, pp. 107–121, 1996.

[12] A. Kleen, *Linux Manual Page: UNIX Domain Sockets*.

[13] J. Mauro and R. McDougall, *Solaris Internals*. Prentice Hall, Oct 2001.

[14] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[15] J. Nieh, J. G. Hanko, J. D. Northcutt, and G. A. Wall, "*SVR4 UNIX scheduler unacceptable for multimedia applications*". In 4th *Int'l Workshop Network & Operating System Support for Digital Audio & Video*, Nov 1993.

[16] J. Nieh and M. S. Lam, "*The design, implementation and evaluation of SMART: a scheduler for multimedia applications*". In 16th *Symp. Operating Systems Principles*, pp. 184–197, Oct 1997.

[17] B. Paul, "*Introduction to the Direct Rendering Infrastructure*". http://dri.sourceforge.net/doc/DRIintro.html, August 2000.

[18] M. A. Rau and E. Smirni, "*Adaptive CPU scheduling policies for mixed multimedia and best-effort workloads*". In *Modeling, Anal. & Simulation of Comput. & Telecomm. Syst.*, 1999.

[19] B. Shneiderman, *Designing the User Interface*. Addison-Wesley, 3rd ed., 1998.

[20] Silicon Graphics Inc., "*OpenGL*". http://www.opengl.org/.

[21] D. A. Solomon and M. E. Russinovich, *Inside Windows 2000*. Microsoft Press, 3rd ed., 2000.

[22] W. R. Stevens, *Advanced Programming in the Unix Environment*. Addison Wesley, June 1993.

[23] W. R. Stevens, *UNIX Network Programming*. Vol. 1. Networking APIs: Sockets and XTI, Prentice Hall PTR, 2nd ed., 1998.

[24] L. Torvalds, A. Cox, and I. Molnar, *Improving Interactivity*. http://kerneltrap.org/node.php?id=603, Mar 2003. Linux Kernal Mailing List, Summarized Thread.

[25] L. Trovalds, A. Cox, and many others, "*The Linux Kernel Sources, Version 2.4.8*". http://www.kernel.org.

[26] C. A. Waldspurger and W. E. Weihl, "*Lottery scheduling: flexible proportional-share resource management*". In *Symp. Operating System Design & Implementation*, November 1994.

[27] X Consortium, "*X Windows System*". www.X.org.

[28] Y. Zhang and A. Sivasubramaniam, "*Scheduling Best-Effort and Real-Time Pipelined Applications on Time-Shared Clusters*". In *ACM Symp. Parallel Algorithms & Architectures*, pp. 209–218, July 2001.