Process Prioritization Using Output Production: Scheduling for Multimedia

YOAV ETSION, DAN TSAFRIR and DROR G. FEITELSON
The Hebrew University

Desktop operating systems such as Windows and Linux base scheduling decisions on CPU consumption, prioritizing processes that consume fewer CPU cycles. The assumption is that interactive processes gain from this approach as they spend most of their time waiting for user input. However, this doesn't work for modern multimedia applications, which require significant CPU resources. We therefore suggest a new metric to identify interactive processes, by explicitly measuring interactions with the user, which can then be used to prioritize processes for scheduling. This was implemented in Linux, and measured using a veriety of applications. The results indicate that it is very effective in distinguishing between competing interactive and non-interactive processes, and emphasize the need for further research on integrating interactivity data into a desktop scheduler.

Categories and Subject Descriptors: D.4.1 [**Process Management**]: Scheduling; H.5.1 [**Multimedia Information Systems**]: ; H.1.2 [**User/Machine Systems**]: Human factors

General Terms: Algorithms, Design, Performance, Human Factors

Additional Key Words and Phrases: Multimedia, Resource management

1. INTRODUCTION

Modern desktop computers are required to run a plethora of different applications: text editors, spell and style checkers, network downloads, playing of audio and video, GUIs with animations, etc. In many cases several threads from different applications execute at once, some in the background and some with direct user interaction. The operating system scheduler is charged with allocating CPU resources to the different threads, with the goal of prioritizing those that are most important to the user.

Prevalent commodity systems use a simple scheduling scheme that has not changed much in 30 years. Processes are scheduled in priority order, where priority is inversely related to CPU usage. CPU usage is forgotten after some time, in order to focus on recent

Part of this work was presented in preliminary form at NOSSDAV 2004 [Etsion et al. 2004].
Yoav Etsion was supported by a Usenix scholastic grant.
Authors' address:
School of Computer Science and Engineering,
The Hebrew University,
91904 Jerusalem, Israel
{etsman,dants,feit}@cs.huji.ac.il

activity instead of distant history. This is true for Windows family [Solomon and Russinovich 2000], Linux [Bovet and Cesati 2001], and other variants of Unix such as Solaris [Mauro and McDougall 2001], AIX, and BSD [McKusick et al. 1997].

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 (interactive) processes. This was sufficient in the past, when user-computer interaction was essentially text-based, and interactive applications exhibited very low CPU consumption. Nowadays, computer workloads (especially on the desktop) contain a significant multimedia component. These workloads are not well supported by conventional operating system schedulers [Nieh et al. 1993; Etsion et al. 2004], as multimedia applications are very demanding in terms of CPU usage and are therefore indistinguishable from traditional background (batch) jobs.

For example, the left graph in Fig. 7 of the experimental results demonstrates what happens when a Xine movie-player displays a short clip along with an increasing number of synthetic 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. Thus, 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%.

To prevent such scenarios, better support for interactive and multimedia jobs is required. We suggest that, on a general purpose system, this be done in two phases. First, the system has to correctly identify the interactive and multimedia processes. Second, the system has to schedule all the running processes, giving special attention to the interactive and multimedia ones.

As an alternative to CPU usage we propose that scheduling decisions be based on a direct measurement of the level of user interaction [Evans et al. 1993]. This is done by monitoring the amount of user I/O performed by the different processes (e.g. mouse and keyboard input events and screen-oriented output events). We also monitor inter-process interactions, to identify the closure of processes that interact with the user indirectly via another process. This approach captures both traditional interactive applications (such as text editors) and modern multimedia applications, which we collectively denote as being *Human Centered* (HuC).

The availability of this information regarding user I/O enables a new type of scheduling: prioritize processes based on I/O production rather than CPU consumption. In other words, instead of equalizing the CPU consumption of all processes, we try to allocate processor shares based on the various processes' interactions with the user, thus favoring HuC processes over non-HuC ones; at the same time, we take care to eliminate the possibility of process starvation. The right side of Fig. 7 shows this for the Xine process competing with the stressors. No matter how many stressors are added, the scheduler correctly identifies the Xine processes and continues to allocate all the required resources to Xine; the stressors have to make do with whatever is left over. This is completely automatic, and requires neither modifications to Xine nor special actions by the user.

Other scenarios, however, can be problematic. For example, what should be done when multiple interactive applications compete against each other? We have experimented with creating a model of how output production depends on CPU usage, and using this to allocate CPU resources so as to equalize I/O production. This works well for competing applications from the same class, e.g. multiple movie viewers, and leads to an equitable and graceful degradation of the service received by all of them. But it might do the wrong thing when an application that produces sporadic text output competes with a graphical visualization. The bottom line is that using output production for prioritization is complex, and probably cannot be used as the sole metric. Instead, it should be integrated with other metrics in order to provide the scheduler with a complete picture of application behavior. However, to fully understand human-computer interaction dynamics in order to fine tune a scheduling algorithm such as the one we propose, more research is needed in the cognitive area.

The rest of this paper is organized as follows: We survey related work in Section 2, then go on to describe our methodology and test platform in Section 3. The next three sections discuss the first phase of HuC scheduling — identifying the HuC processes. Section 4 examines the failure of the standard identification based on CPU consumption patterns. Section 5 explains the concepts of I/O quantification, while Section 6 describe how this is measured and used to identify HuC process. The second phase, scheduling the HuC processes per-se, is discussed in Section 7, and its integration into the Linux kernel is described in Section 8. We then show experimental results comparing the classical CPU-based scheduler with our HuC scheduler in Section 9, and conclude in Section 10.

2. RELATED WORK

Scheduling on a desktop machine attempts to achieve a combination of goals. One is to run interactive applications in a timely manner, providing low response times. Another is to enable the use of leftover processing resources for background processes, be they non-interactive jobs belonging to the machine's owner (e.g. a large compilation or download), or imported work as part of a load sharing environment. The challenge is that this should not interfere with the support for the interactive jobs.

Traditionally, schedulers on desktop machines prioritized processes based on their CPU usage, or rather, lack of CPU usage. However, the reasoning that lack of CPU usage identifies interactive processes is now obsolete [Etsion et al. 2004; Nieh et al. 1993]. Modern desktop applications span a whole spectrum of CPU usage levels, from text editors that use little CPU up to interactive games that can dominate 100% of the CPU. Those that use significant CPU resources are therefore indistinguishable from non-interactive jobs such as compute-bound computations or large compilations. Thus a scheduler based on CPU usage patterns will not give the interactive applications sufficient resources, leading to degraded performance or an inability to run a mix of interactive and non-interactive jobs.

Desktop operating systems such as Windows have taken initial steps to identify and prioritize interactive applications, e.g. by increasing the CPU allocation of threads that are associated with the focus window or have waited for a slow device like the keyboard [Solomon and Russinovich 2000]. However, this is oblivious to their actual needs, and does not necessarily solve the problem (what about displaying output in a non-focus window? or if the focus application depends on services provided by others?).

Several research projects have devised systems specifically to allow interactive multimedia applications to run successfully. These can be broadly classified into two groups, that place the burden on the programmer or on the user of the application.

The solution adopted by the first group is to provide soft real-time support so that multimedia applications can sustain frame rates and audio sample rates. The programmer must then use special interfaces to utilize these services. On the system side, support includes three components: high resolution timing services, a preemptive and responsive kernel, and appropriate scheduling [Goel et al. 2002]. Several schedulers have been designed and implemented, including SMART [Nieh and Lam 1997] and BEST [Banachowski and Brandt 2002]. The latter has the distinction of also trying to identify applications with periodic computation needs automatically.

The solution adopted by the second group is to use fair-share scheduling [Childs and Ingram 2001]. This does not require any modifications in the applications, but shifts the burden of configuring the system to the user, who must specify the resource requirements of select applications. This is probably not a good solution for transient interactive and multimedia tasks that come and go during normal work. Example systems of this type include Lottery Scheduling [Waldspurger and Weihl 1994] and Borrowed Virtual Time [Duda and Cheriton 1999]. An extension to this is the use of hierarchical schedulers, that allocate CPU time between other, class-specific schedulers [Goyal et al. 1996; Candea and Jones 1998]. This principle is somewhat similar to the hierarchical scheduler we describe in Section 8. The Eclipse operating system [Bruno et al. 1998] takes an additional step, and supports guaranteed portions of multiple resources at once: not only the CPU, but also memory blocks, disk bandwidth, and network bandwidth.

In a related vein, Zhang and Sivasubramaniam [2001] 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. Similar work by Rau and Smirni [1999] requires the user to specify a tolerance threshold for performance degradation, and the system then adjusts allocations to try and meet this specification. Their notion of quality for multimedia applications is based on missed deadlines, which is closely related to our use of output rate (a generalization of frame rate).

In contrast to the aforementioned related work, one of our principal goals is to automate the scheduling mechanism, shifting the tuning burden from the programmer/user to the scheduler itself, by making it aware of user–process interaction.

3. EXPERIMENTAL METHODOLOGY

Before presenting our arguments and results, we first describe our platform and introduce the applications used to evaluate the newly proposed scheduler.

3.1 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 was 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 new Linux 2.6 kernel, and is more suitable for multimedia applications which require millisecond timing resolution [Nieh and Lam 1997; Etsion et al. 2003]. We have also verified that the increase in overhead is negligible [Etsion et al. 2003].

3.2 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 sizable buffer in memory (typically 4MB), and only exported at large intervals. This export is performed by a dæmon 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 $< 1\%$. 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).

3.3 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. These serve as two variants of background, that can absorb any number of available CPU cycles, and compete with HuC processes. They differ however in their I/O behavior: while stressors represent completely CPU-bound applications, kernel compilation also employ massive disk I/O.

—**Movie players**: MPlayer and the Xine MPEG viewer, which were used to show various video segments encoded with different standard frame rates. While MPlayer is a single threaded application, Xine's implementation is multithreaded, making it a suitable representative of this growing class of applications [Flautner et al. 2000]. In our experiments audio output was disabled, to allow focus on interactions with the X server.

—**Modern interactive applications**: The Quake III Arena action 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 dæmons. Of these, the most important with regard to interactive processes is obviously the X server.

4. THE FAILURE TO IDENTIFY HUC PROCESSES BY CPU USAGE PATTERNS

Prioritization based on CPU usage can take various forms. In this section we show that all of them do not work, as modern HuC processes may use significant CPU resources, and are essentially indistinguishable from non-HuC work.

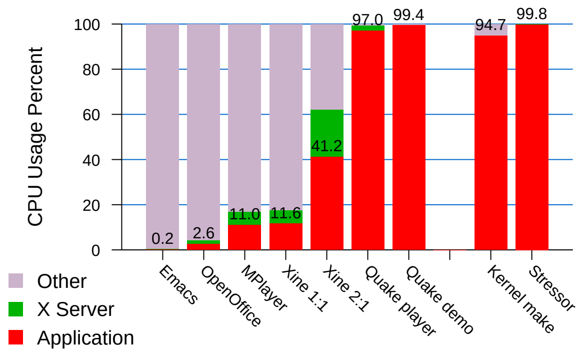


Fig. 1. *CPU consumption of different applications expressed as a percentage of the wallclock time (from [Etsion et al. 2004]). Xine 2:1 means resizing to 200% (quadruple the area).*

4.1 CPU Consumption

The simplest measure of CPU usage is total consumption. Most general purpose schedulers base priority mainly on this metric. Processes that use the CPU lose priority, while those that wait in the queue gain priority.

The question, however, is whether low CPU consumption can be used to identify HuC processes. Figure 1 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.

4.2 Effective Quantum Lengths

While CPU consumption is the main metric used by current schedulers, other (new) metrics are also possible. A promising candidate is the distribution of *effective quantum lengths*. An effective quantum is defined to be the time period between the time a process is allocated a processor and until the processor is relinquished, either because the process is preempted when its allocation expires or when a higher priority process awakens, or because the process blocks, waiting for some event. 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, and because they often need to use timer alarms to pace themselves (e.g. to generate the correct frame rate regardless of processor speed). Thus we expect to see a difference between the allocated quanta and the effective ones in HuC processes, but expect non-HuC processes to typically use their full allocation.

Figure 2 shows these distributions for different groups of applications. Multimedia applications, in particular, are indistinguishable from other application types: on one hand

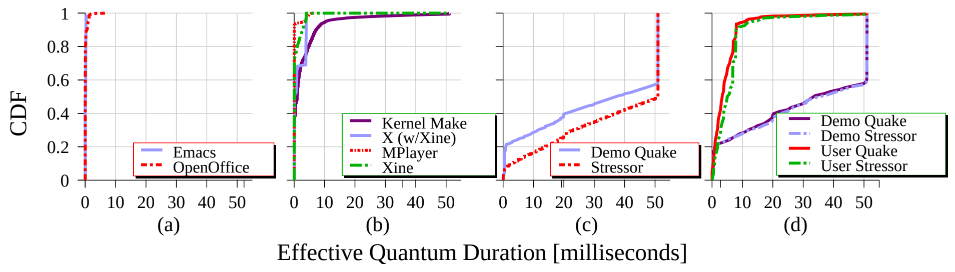


Fig. 2. *Cumulative distribution function of the effective quanta when applications are run alone.* **(a)** *Editors have very short effective quanta.* **(b)** *Movie players also have short effective quanta, but this is similar to the profile of the kernel-make batch job.* **(c)** *Quake can consume all available CPU cycles, so when running in demo mode it behaves like a stressor. Both are occasionally interrupted by various system dæmons, causing around 50% of the effective quanta to end prematurely.* **(d)** *When a stressor runs together with Quake (in either demo or user mode), both end up with the same distribution, because Quake interrupts the stressor.*

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

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.

4.3 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 and timing devices, through which they communicate with the user in a paced manner. 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 induced 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 I indicate that this new metric also fails to make a clear distinction between HuC and other processes. Quake is again similar to stressors, and Xine looks like kernel-make.

5. QUANTIFYING USER I/O

Before describing mechanisms to track interactions between processes and the user, we must first define what we want to track and how. Simply put — how do we quantify user interactions?

5.1 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, can be argued to convey about 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 process receiving the most recent user input should get the highest priority.

Reflecting these considerations, we implement input ratings as a binary state variable: either the process has received input recently, or it has not. "Recently" means within a certain predefined number of seconds, which is a tunable parameter (see below).

It should be noted that this approach has the additional desirable effect of implicitly recognizing the application associated with the focus window. X assigns input to the focus application, so input events imply focus. Thus we do not need a separate mechanism for prioritizing the focus application, as is done in Windows. It can also be argued that our approach is even better, as it uses real input (a direct measure) rather than the focus property (an indirect measure).

5.2 Quantifying Output to the User

Quantifying output is more complex than input: firstly, because various applications may simultaneously produce output to different windows, secondly, because of different output modalities (e.g. with or without the concept of a frame rate), and finally, because we don't know which of these output events is more significant to the user.

Two metrics suggest themselves for measuring importance of output to different windows: the size of the window, and the rate of change. We prefer to use rate, motivated by the fact that human vision is known to be more sensitive to movement (a remnant of our hunting predecessors) [Shneiderman 1998]. Thus quantifying the rate of change produced by each application will lead to a reasonable guess about which process has the user's attention.

The question remains of how to quantify the rate of screen changes. We see three possible candidates.

(1) The simplest approach is to count output events. This may be justifiable in cases where each output event represents a unit of information, such as printing a single character. However, in general output events may come in very different sizes. Specifically, this approach is not suitable for the X-Windows system, as an X-protocol output request can change a single character, draw a line, or change the entire image in a window.
(2) Another option is to count pixels. Thus a "large" output event that modifies many pixels will confer a larger amount of user interaction, in accordance with the notion that human vision is more sensitive to movement.
(3) The third option is to use normalized pixel counts, based on the following formula:

$$\text{output rate} = \frac{\text{pixels changed in last second}}{\text{window size in pixels}}$$

Normalizing the changed-pixel count by the window size is motivated by the desire to distinguish between raw I/O and a higher level of I/O. Consider video or animation as an example. When counting the output in pixels the result depends on the window size, which is not considered by the application as part of its quality of service. But normalizing by the window size yields a count of the frame rate, which is independent of window size, uses the same quality metric as the application level, and places the competing processes on an equal footing.

Our prototype implementation uses the third approach, as it is implicitly geared toward video applications.

6. IDENTIFYING HUC PROCESSES BASED ON USER INTERACTION

Our basic idea is 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 6.1, 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 6.2, that tracks interprocess communication to unearth dependence relationships between them. Finally, newly forked processes inherit the HuC counts of their parent. Together, these mechanisms allow the scheduler to correctly identify and prioritize HuC processes.

Interestingly, the mechanisms described above have been proposed in the past for other uses. The idea of identifying HuC processes as those that interact with the X server, and notifying the kernel about them, was proposed a long time ago by Evans et al. [1993]. However, they did not consider interactions among processes. Using the closure of processes that interact with the X server has been proposed by Flautner et al. [2000] — but in the context of power management, not scheduling.

6.1 Monitoring Direct User I/O

6.1.1 *HuC Devices and the X Server.* I/O between the user and the various processes is mediated by peripheral devices. Identification of user interaction must therefore start with the devices that represent the user: the keyboard, mouse, screen, joystick, sound card, tablets, and touchscreens (to name a few) — which will be referred to collectively as *HuC devices*. For the purpose of this research we've decided to only monitor the "bare necessities", namely the keyboard, mouse, and screen. Furthermore, we focus on the use of windowing systems, and ignore the possible direct use of a text console interface.

Unix environments use the X-Windows system [X Consortium] as the conventional mechanism to multiplex I/O between the user and the various applications. Applications that wish to use the keyboard, mouse, and screen 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 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 [Etsion et al. 2004]. A similar approach can be implemented for the Windows family of operating systems using the *DirectX* subsystem [Gray 2003].

Additional reasons for focusing on X are that it represents the common denominator of all systems (many don't have joysticks, and some even don't have sound cards). Moreover, the applications using other devices are typically the same ones using X: when playing a

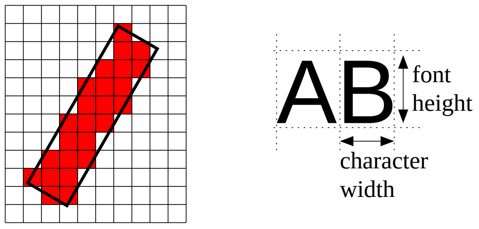


Fig. 3. *Estimation of the area of a diagonal line and a text character.*

game using the joystick, for example, the game displays graphics on the screen and uses the sound card for sound effects. Also, most systems do not allow more than one application at a time to use HuC devices. Thus monitoring other devices will add some information about additional I/O modes, but not about other processes. Finally, trying to incorporate other devices would increase complexity by requiring us to quantify their I/O rates using a common metric, and combine them into a single number.

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) [Paul 2000] is the dominant alternative since it is used by the *OpenGL* graphical library, which in turn is heavily used by graphical software, and in particular games. DRI interacts directly with the graphics controller, circumventing the X protocol. Thus a complete implementation of our ideas should include instrumentation of OpenGL, similar to our instrumentation of the X server described below.

The mechanism described in this section is but one example of implementing user interaction monitoring in a kernel subsystem. The same approach described here for the graphics kernel/user subsystem can be generalized into a scheduler hints mechanism integrated into other major kernel device subsystems, such as *ALSA* for audio, *Input Core* for input devices, and the *Video4Linux* subsystem [Bovet and Cesati 2001; Corbet et al. 2005] (or even the *DirectX* subsystem in Windows [Gray 2003]).

6.1.2 *Instrumenting the X Server.* The mechanisms of Section 5 were implemented by instrumenting the X server. The code for handling process input is 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. Counting changed pixels is also feasible since the X protocol defines a reduced set of only seventeen graphical X-requests that are available to clients (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, using a simple bounding box schema. Figure 3 depicts how this works for two common operations: drawing a diagonal line, and drawing a character [Etsion et al. 2004].

Note, however, that output events may refer to hidden portions of windows. As our motivation for using rate of change as a metric is based on how change is perceived by the user, changes that can't be seen by the user shouldn't be included in the application's ratings. We therefore hooked into the X clipping mechanism in order to find out how much of the change is indeed visible to the user.

To be useful, the data regarding each I/O event needs to be attributed to the correct process and communicated to the kernel. The X server maintains in its internal data structures a *client record* for each client. We have added three fields to this record:

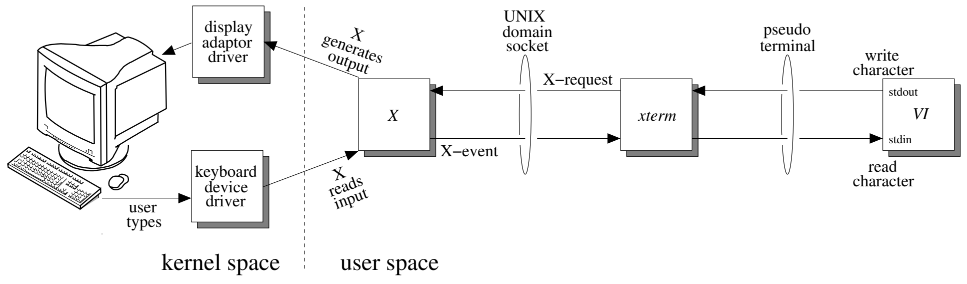


Fig. 4. *Information flow between the user and the VI text editor.*

—client's process ID (pid),
—client's input status (input or not), and
—client's output ratings (the relative pixel change rate)

The values of these fields are communicated periodically to the kernel using the X native timer mechanism (the prototype does this once a second) through the standard POSIX *sched_setparam* system call [Gallmeister 1995], to which we have added a few non-standard parameters. In addition to these periodic updates, 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 – for example, raise the priority of a process that just received some input.

Client pids are needed because eventually the scheduler will base its decisions upon the I/O ratings associated with each process. X doesn't maintain pids, because one of its major design goals is to serve local or remote clients in the same way, and the pid of a remote client is meaningless. 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 paper). 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, and non-standard Unix-domain socket options implemented in Linux provide access to the sender's pid.

6.2 Indirect User I/O

As noted above, the main process that interacts with the user in a Unix system is the X server. HuC applications interact with the user indirectly, using X and other processes as intermediaries. Figure 4 demonstrates this with a scenario in which the user writes some document using the *VI* text editor from within an *xterm* terminal 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 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 to include processes that indirectly interact with the user.

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.

6.2.1 *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. Finding all the dependencies is therefore difficult [Mosberger and Peterson 1996].

The alternative is to find a single mechanism that facilitates an *approximation* of the interactions that have taken place. For this, we propose to monitor attempted insertions into the ready queue. When one process causes another to enter the ready 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 queue are always performed via the *try_to_wake_up(process)* function. Significantly, the invoking process does not verify that the target process is indeed waiting (which explains the "try" prefix in its name). Thus, an invocation of *try_to_wake_up* represents a true logical dependency between the two processes, regardless of what their current status happens to be. A similar idea was recently explored by leading Linux developers [Torvalds et al. 2003], but eventually was not adopted in the 2.6 kernel.

The above heuristic has the apparent drawback that some dependencies might go unnoticed. This can happen when non-blocking mechanisms are used, e.g. if information is passed using shared memory. However, using shared memory is typically accompanied by some synchronization mechanism such as semaphores, which do include blocking. In addition, we consider sets of processes that share their address space as a single entity, rather than considering each of them individually. As a consequence, we find that in practice our heuristic produces excellent results in identifying all the processes in the X server's IPC graph closure.

The interprocess communication graph is by far the most complex part of the HuC system. We implemented it in full to get the most accurate user I/O statistics, by which we can prioritize the various processes. While this is befitting for a proof-of-concept implementation, it is less desirable to incorporate it in a production system. However, it is sufficient to approximate this graph using mechanisms smilar to those used to support priority inheritance and overcome priority inversion issues, common in modern operating systems [Silberschatz et al. 2004; Solomon and Russinovich 2000; Mauro and McDougall 2001].

6.2.2 *Propagating HuC Input.* As noted above, input is a direct reflection of user interest. We therefore define the "HuC input" status to be infectious. This means that if a "HuC input" process inserts another process into the ready queue, that process also becomes "HuC input". Moreover, this is communicated to the kernel immediately, without waiting for the next periodic update.

For example, consider the scenario depicted in Fig. 4. The X server is identified as HuC input when it reads a character from the keyboard. When the character is passed to the xterm application, xterm too becomes HuC input. When it is passed to VI, so does VI.

6.2.3 *Propagating Output Ratings.* Output is different from input in that it is quantified rather than being binary. The output ratings need to be propagated in the opposite direction from process interactions, in order to assign the ratings to the processes that indeed initiated the output operation.

Process interactions induce a directed graph, which we call the *Process Dependency Graph* (PDG). The nodes of this graph are all active processes in the system. The graph contains an edge (P_i, P_j) iff P_j was recently inserted to the ready queue due to an action taken by P_i. Returning to the example in Fig. 4, when VI generates output and sends it to xterm, we get $(P_{VI}, P_{xterm}) \in PDG$; then we also get $(P_{xterm}, P_{Xserver}) \in PDG$ when xterm forwards the request to X.

Our instrumentation of the X server includes the quantification of output attributed to processes that interact with it directly, e.g. the xterm process. The PDG is used to propagate this rating further. In the example, the edge (P_{VI}, P_{xterm}) implies that xterm depends on VI, and that VI may therefore be the source of the output. So the output rating of xterm is also attributed to VI. If several such edges exist, the output rating is divided among them according to their weights. The weights reflect the number of times that the process at the head of the edge tried to wake the process at its tail.

In future work we intend to consider possible simplifications of this approach. Specifically, it may be better to propagate output information on the fly as each interaction occurs, as is done for input.

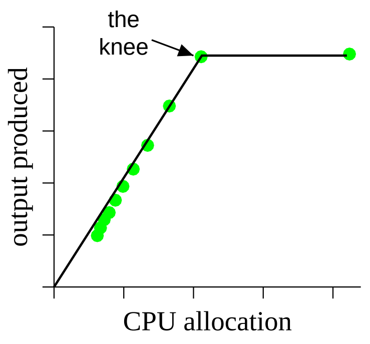
6.2.4 *Aging the Data.* Applications may change their behavior over time, e.g. accept input parameters from the user interactively and then perform a long non-interactive computation. It is therefore desirable that the identification of HuC processes be based only on recent user I/O. In order to actually maintain the I/O data, we need to define the meaning of “recent”.

After some deliberations, our final algorithm is very simple: we have a tunable parameter that specifies for how long data is maintained, currently set to 8 seconds. Initially we experimented with exponential aging, in which the weights of the edges are divided by a factor of 2 each second, until they become smaller than one. However, edge weights are typically smaller than 100, so this implies that output data is retained for 5–7 seconds. Also, aging input data would need a different treatment, as it is binary to begin with. In light of these considerations, using a life span of 8 seconds is a reasonable compromise that provides for even handling of both data types. Any event on either an input or output edge initializes its life time. While this seems to work nicely in practice, further experimentation with real users is required to fully justify this approach, or to refine it.

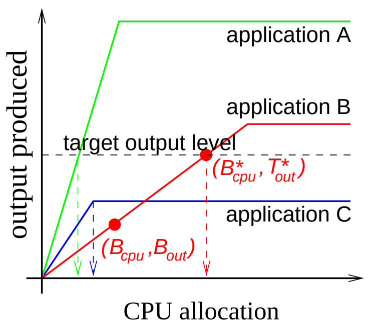
7. THE HUC SCHEDULING ALGORITHM

Until now, we have only discussed how to identify the HuC processes. Now we describe the HuC scheduling algorithm — how we allocate CPU time for these and competing processes.

Given the approximation of how much user I/O is associated with each process, we need to use this information to decide on CPU allocations. The traditional Unix scheduler employs a negative feedback mechanism to achieve a uniform distribution of CPU resources (within the constraint that some processes may not need as much as others). When a process runs its priority drops, until the CPU is relinquished and given to another process. Our HuC scheduler, by contradistinction, allocates CPU resources so as to achieve *uniform output rates* by the different processes. This means that CPU allocations need not be equal; rather, each process gets the CPU resources it needs to produce the target level of output. In this sense it is similar to the schedulers proposed by Massalin and Pu [1990]



(a) The knee model of application behavior: I/O activity is linearly dependent on CPU allocation, up to the maximum needed. Data points are measurements of Xine.



(b) Using the knee model allows for an estimation of what CPU allocation will lead to a desired level of output production.

Fig. 5. The Knee model for application behavior, and its use in our scheduling algorithm

and by Steere et al. [1999], who adjust the processing power allocated to a producer and a consumer so that the producer's production matches the consumer's consumption.

The scheduler operates at two levels of granularity. Like the Linux scheduler, we have a notion of *epochs*. In each epoch, an *allocation* is made for each process. During the epoch the different processes execute in an interleaved manner as decided by the *dispatch* algorithm. When all processes have exhausted their allocations, or no ready process is available, a new epoch is started. Unlike Linux, we set a maximum on the duration of an epoch, 200ms in the current implementation. This is needed in order to bound the time until a process gets to run. The scheduler guarantees that every process will get some CPU time at least once per epoch. Note that until recently $\sim$ 200ms was the default length of a scheduling quantum on many Unix variants, but in practice much smaller effective quanta are typical for many interactive applications on today's hardware [Etsion et al. 2003].

7.1 The Knee Model of Application Behavior

In order to achieve a certain target level of output activity, we need a model of how output production is related to CPU resources. We suggest the *knee model* of application behavior for this purpose. Simply put, it states that the level of output is linearly related to the CPU allocation, up to a certain limit. Above that limit the application does not need any more CPU resources, so additional allocations will not be used, and in particular, will not lead to additional output.

Actual measurements of application behavior support this model. Fig. 5(a) shows such measurements for the Xine MPEG viewer (measurements for the X server exhibit a similar structure). The measurements were conducted by running X and xine with different numbers of competing stressor (synthetic CPU-bound) processes under the default Linux scheduler. When there are no stressors, the application simply uses whatever it needs (we consider the "allocation" to include the system's idle time, since it is in fact at the application's disposal); this serves to identify the knee point. When enough stressors are present, the application receives less CPU resources, and produces less output.

Importantly, the model is simple enough so that model parameters are accessible to the scheduler. At runtime, the scheduler can keep track of how much CPU was actually used by each process, and also get a quantification of the output produced (as described in Section 6). The quotient of the output volume to the CPU usage provides the slope of the line. Any discrepancy between the allocated and used CPU gives an indication about the location of the knee.

7.2 CPU Time Allocation

If the total CPU requirements of all applications are less than the full capacity, the scheduler does not have to make any hard decisions, allowing all application to reach their desired output rate. But if requirements exceed capacity, the scheduler needs to decide how to allocate the CPU resources. Our algorithm performs this allocation in a way that will lead to uniform levels of output production, subject to the constraint that some applications can only produce a limited amount of output. This is similar in spirit to a fair-share scheduling algorithm, except that the shares are calculated automatically (and indeed this prioritization mechanism can be used on top of most fair-share schedulers).

Let us start by justifying this approach. Consider a system that is used for video-conferencing, and is displaying two incoming streams, one in a small window and one in a larger window, with 4 times the area. Assume also that the system is overloaded, so the processes displaying the two streams cannot get the full CPU resources they need to display their respective streams at the full frame rate. Using a conventional scheduler, both processes will receive about the same CPU time. But displaying a frame in the large window requires about 4 times more processing power than displaying a frame in a small window [Etsion et al. 2004]. As a result, the smaller window, which is probably less important (otherwise, why did the user choose to make it smaller?) will end up displaying 4 times more frames, and providing better video quality! At the same time, the larger, more important window, will lose more frames and provide reduced quality.

Now consider allocating CPU time so as to achieve a desired output rate. Recall that we define the output rate in relative terms, that is counting pixels that changed divided by total pixels in the window. Using this metric, having equal output rates translates to displaying the videos at the same frame rate, regardless of window size. So an allocation that equalizes output rate achieves the desired balance between competing applications. Under this allocation, the small-window Xine will receive only a quarter of the CPU time that the large-window one gets.

Our allocation algorithm is based on the knee model as illustrated in Fig. 5(b). As the model is simply a linear relationship, the allocation is just the amount of CPU that will generate the desired level of output. For example, assume that for application B the current CPU allocation was B_{cpu} and the generated output was B_{out}. If the target output level is T^*_{out}, the allocation B^*_{cpu} will be

$$B^*_{cpu} = \frac{B_{cpu}}{B_{out}} T^*_{out}$$

because (B_{cpu}, B_{out}) and (B^*_{cpu}, T^*_{out}) lie on the same slope, and therefore have the same ratios. For application C the maximal possible output rate is lower than the target, so the knee has to be identified and the allocation will be that of the knee. Note that when the system is underloaded (i.e. requirements are less than capacity) this is the case for all applications, and they all get allocations that bring them to their perspective knees.

One may wonder at this point how the target value of T^*_{out} is set. The answer is that there is a circular dependency, in that T^*_{out} should be set so that the total CPU time allocated will be about the duration of the scheduling epoch. To break the circle we initially set T^*_{out} to be the average of the $A_{out}, B_{out}, \ldots$ values of all HuC processes from the previous epoch, derive CPU allocations, and then rescale them so that their sum matches the total allocation. If this sum is less then the total CPU allocation, no scaling is needed — meaning the system is not overloaded and all applications are granted their desired CPU share.

A special case occurs when a new process is inducted into the HuC class. Such a process might have had a small allocation previously, but a very large new allocation, especially if its slope in the knee model is low. However, it is dangerous for system stability to give a large allocation at once to such a new process. The solution is to grow exponentially. At each new allocation, the process is limited to some factor α times the previously used time, e.g. doubling it ($\alpha = 2$). The expression for the CPU allocation is then

$$B^*_{cpu} = \min\left\{\frac{B_{cpu}}{B_{out}}\,T^*_{out},\ \alpha B_{cpu}\right\}$$

For new HuC input processes, which do not yet have any measured output, an arbitrary initial allocation is used (in the prototype, this is 1% of the epoch). The reason for this is that when a process receives input we know it is important, but we do not yet know how much CPU it needs. Therefore we initially just give it a chance to produce output, and base future allocations on this output.

Another special case concerns the X server. As all I/O activity passes through X, it needs to be able to handle a larger amount of I/O than T^*_{out}. We therefore set its allocation based on T^*_{out} multiplied by the number of applications using it.

7.3 Dispatching

Dispatching is the decision of which process to run next, given that a few ready processes with non-zero allocations are available. In principle, we would like all processes to make progress together, at their respective rates (i.e. according to their allocations). In practice, progress is made in a granular manner, as only one process actually runs at any given time. The challenge is then to select the processes in a way that will allow all of them to make progress at about the correct rate, without leaving any process too far behind (which amounts to starvation).

The common solution to this problem is to schedule according to *virtual time* (VT) [Nieh et al. 2001]. Each process has a virtual time, that advances at a rate that depends on its allocation (or weight). Thus, the virtual time of a process that has a large allocation will advance more slowly when it runs, and the virtual time of a process with a small allocation will advance more quickly. When called, the scheduler chooses the process with the smallest virtual time to run next.

For example, consider four processes of which one has an allocation of 4 "ticks"[1] (call it process A) and the other three an allocation of 1 each (call them B, C, and D). When B, C, or D run, their VT immediately becomes 1. But when process A runs, its VT only increases by $\frac{1}{4}$. It will therefore have an advantage over those processes whose VT is already 1.

[1]The units of time allocation, defined by the operating system clock interrupt rate.

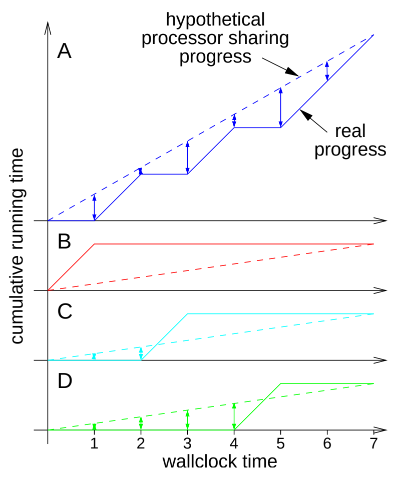


Fig. 6. *Using PSVT to dispatch 4 competing processes. Initially B happens to run. As a result A, which has the largest allocation, suffers the largest deviation from its PS progress (vertical arrows at time 1). After running A for a quantum this is no longer the case, and C is selected (D could have been as well; arrows at time 2). After running C, A is again more behind than D, so it runs again, and D is delayed again (arrows at time 3). At time 4, D is finally selected. The end result is that the symmetry between the low-allocation processes is broken, and they are dispersed rather than running one after the other.*

The problem with VT is that once process A runs, its VT ($\frac{1}{4}$) is already higher than that of any process that has not run yet (0). Therefore the scheduler will select B, C, and D within the first 4 time slots, and leave the last three slots for A. To solve such problems, one can consider the upcoming run time in the decision, and use the *virtual finish time* (VFT) instead of the VT. Thus we hypothetically add the next quantum to each process in turn, and see what its virtual time will be if the quantum is allocated to this process.

Note that when some allocations are very small, VFT may be as bad as VT: it simply delays B, C, and D to the end rather than running them first. However, interactive and multimedia processes typically fragment their allocations into many short runs (less than a full tick), based on their real-time needs [Etsion et al. 2003]. It is therefore actually better to use VT and not VFT, because VT better reflects how CPU time is really used. VFT will lose this information, as it always assumes that the full allocation will be used.

What we would really like to achieve is an interleaving of the different processes, ideally a sequence of dispatch decisions like A, B, A, C, A, D, A. This can be approximated by a scheme we call *processor sharing virtual time* (PSVT). Under this scheme, scheduling priority is proportional to the difference between a process's running time and its hypothetical running time if the system were to use processor sharing. Under processor sharing, all processes advance all the time, with rates that are proportional to their shares (dashed lines in Fig. 6). Thus when process A does not run, its PS runtime advances quickly, while its real runtime stays the same, leading to a large mismatch and a high priority. As a result, the scheduler will tend to pick process A after it has not run, even if some low-allocation process has not run at all yet.

The current implementation supports both VF and VFT dispatching. We plan to implement PSVT in the future.

7.4 Workload Dependence

As hinted above, some of the choices made in designing the algorithm are workload dependent. For example, the idea of equalizing output rates as measured by normalized counts of pixel modification is geared toward multimedia applications, allowing the same frame rate to be achieved. But this may be the wrong thing to do in other scenarios. For example, consider the following.

—Some applications simply do not have a frame rate. For example, if a kernel make competes with a movie player, the sporadic prints from the make will create much less output, and therefore the make will be given a much larger allocation in order to enable it to make up.
—Just as multimedia applications exhibit CPU-usage profiles that are indistinguishable from those of compute-bound processes, so do animated popup ads display I/O profiles that are indistinguishable from multimedia applications. By prioritizing I/O we also prioritize such popups. This however is not unique to the HuC scheduler, as this phenomenon also affects CPU pattern based schedulers. For example, a web browser experiencing popups under the regular Linux scheduler will consume more CPU thus having its priority reduced.

The lesson from this is that there is probably no single solution that will be good for all possible situations. However, it seems that monitoring I/O is certainly an interesting addition to the toolbox of system designers. In some cases, it provides exactly the support that is needed. In others, it may be possible to combine it with other tools to achieve the desired results.

8. INTEGRATION WITH LINUX

The HuC scheduling algorithm described above specifies how CPU resources are allocated to HuC processes. But a general purpose desktop system also runs other types of processes. Moreover, processes may be classified as belonging to different classes during their execution. In this section we describe the allocation of CPU resources between the different classes, and the issue of class mobility.

8.1 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 [Gallmeister 1995]). 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 way that will allow new policies to be easily incorporated. Our design was inspired by that of the Solaris 8 scheduling scheme [Mauro and McDougall 2001] 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 goes to the top class with a ready process, and “asks” it to pick its most desirable process.

Class	Subclass	Description
Realtime	FIFO	POSIX first-in first-out
	RR	POSIX round-robin
	KTHREAD	kernel threads
Collective	HUC	processes identified as HuC
	OTHER	Linux default
Idle	IDLE	idle loop

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

Table II lists the scheduling classes we have implemented in our scheduler. FIFO and RR are retained with the same semantics as in Linux. KTHREAD is populated by the various kernel processes which may be considered as part of the operating system (e.g. dæmons involved with paging). Originally such processes belonged to the OTHER class. But once we identify HuC processes and give them a higher priority, we need to ensure that we do not starve these system processes, as this may have a disastrous effect on the system. We therefore created the KTHREAD class, above the HUC class, but still below the FIFO and RR classes. Prioritizing within the KTHREAD class is done as in the original OTHER class.

Next come all the processes that should share available resources and all make progress collectively. Our scheduler's goal is that HuC processes should be prioritized relative to other processes. The HUC class includes all processes identified as HuC by virtue of having positive input or output ratings. Scheduling within the HUC class is as described in the previous section.

OTHER processes are scheduled as in standard Linux: each has its allocation, and the sum of allocations define the epoch. But this does not necessarily correspond to the epoch as defined in Section 7. To make ends meet, we rescale the Linux epoch so as to fit into our epoch (200ms) after subtracting the allocations to the HuC processes.

IDLE currently contains only the idle loop. However, it is possible to envision situations in which this class will be used to run processes that should only be run when the system doesn't have anything better to do. For example, one can have a TUNE subclass for special system processes that perform self tuning [Feitelson and Naaman 1999], and a STANDBY class for user processes such as participation in the SETI@home effort.

8.2 Class Allocations and Class Mobility

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 the potential danger of an unstable positive feedback loop: processes that generate output get a higher allocation, which allows them to run more, potentially creating even more output (obviously input cannot be affected by a positive feedback loop since it solely 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. This may not to be a problem for new processes that inherit the user-interaction counts of their parents, but it could in principle happen to processes forked by non-HuC processes or processes that change their nature over time. We therefore need a solution that allocates CPU time to processes despite the fact that their I/O ratings are low or nil. Luckily, this meshes in

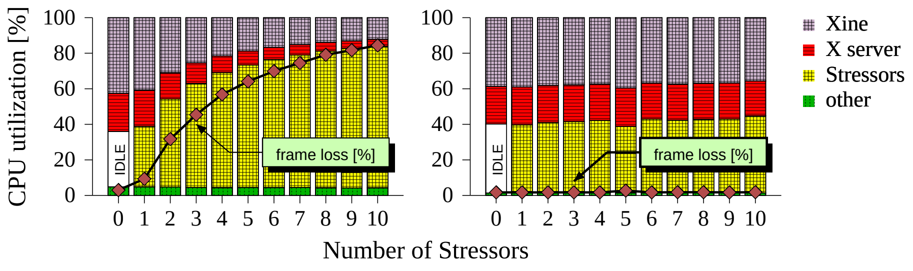


Fig. 7. *Competition between the Xine movie player and background stressor processes. Using the Linux scheduler (left), Xine receives less CPU resources as more stressors are added, resulting in increased frame loss rates. With the HuC scheduler (right) it receives enough resources despite this competition.*

nicely with the concept of an epoch. During an epoch, all active processes get a chance to run. Rather than defining the epochs within each scheduling class, we can define an epoch to span all collective classes.

The allocation of time within the epoch depends on the class. HuC processes get as much of the allocation as they need to meet the target output level. OTHER processes get whatever is left over. Note that if the HuC processes have high requirements, they will tend to monopolize the full epoch; time will be left over only if all HuC processes reach their knee. If not enough time is left for the OTHER class, the epoch is rescaled so as to ensure that each OTHER process gets at least one tick. This is needed to ensure that such processes will be able to generate some output and thus become identified as HuC. And indeed, our tests indicate that starvation is not a problem, and processes that generate output are quickly identified and prioritized.

It should be noted that our approach is only one of a whole spectrum of possibilities. It is also possible to decide to give the OTHER class a certain share of the CPU time, or a share that depends on the number of processes in it. For example, this may be desirable in order to guarantee good progress for background tasks (e.g. compilations or network downloads) even when the user is engaged in an interactive game to pass the time. Evaluating such options is currently left for future work, while the current implementation simply favors the HUC class.

9. 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 one or more HuC process, and different numbers of stressor processes that compete for the CPU. Results here are slightly different from the preliminary ones in [Etsion et al. 2004] due to further development of the scheduler.

9.1 Prioritizing HuC Processes

A striking result is shown in Fig. 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. But with the HuC scheduler Xine and X are identified and given

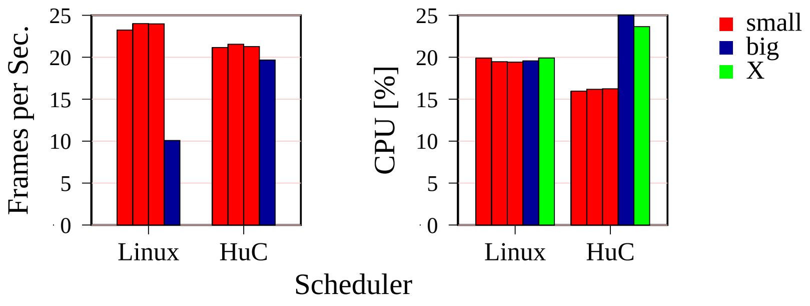


Fig. 8. *CPU allocations and frame rates of competing Xine viewers with different sizes. Small is 1:1, and big is 2:1, in all cases showing the same movie coded with 30 frames per second.*

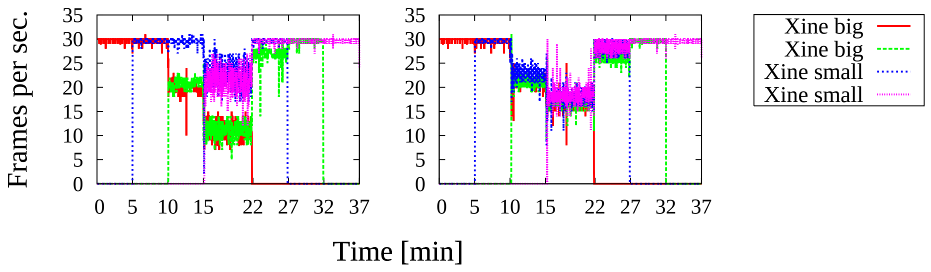


Fig. 9. *Allocations to a dynamic workload of Xine movie players. Four Xine players of different sizes are started 5 minutes apart and run for 22 minutes each. Left: Linux scheduler. Right: HuC scheduler.*

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 about 1% of the CPU resources, and gets it even under the default scheduler; the HuC scheduler provides the same.

9.2 Equalizing Output Production

The above experiments show that HuC processes are correctly prioritized relative to non-HuC processes. But what happens when multiple HuC processes compete against each other? As an example, we test the performance of 4 Xine movie players showing a 22-minute movie at two different sizes. The results are that the Linux scheduler attempts to provide them with equitable CPU resources, allowing the small ones to display many more frames (Fig. 8). Under HuC, on the other hand, the average frame rates are equalized. To achieve this, about 3% of CPU time is taken from each of the three small Xines, and given to the large one.

Another important question is how the allocation adjusts to dynamic load conditions. To check this we again measure four Xines, starting them up at 5 minute intervals. The above behavior was repeated, with the HuC scheduler adjusting allocations so they all achieve the best rate possible at each instant (Fig. 9 right), whereas under Linux the small window sizes are given priority when the system is overloaded (left). For example, in the period

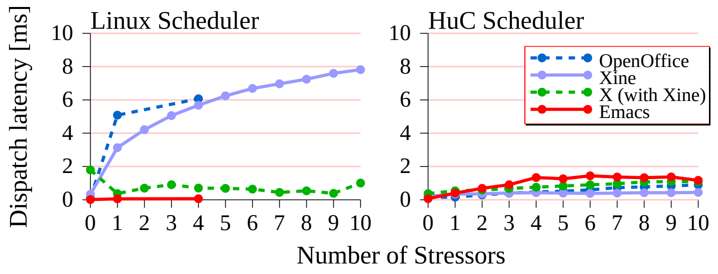


Fig. 10. *Average dispatch latency of HuC applications under the default and HuC schedulers.*

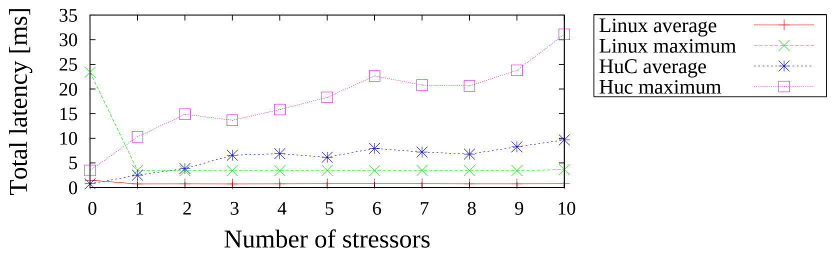


Fig. 11. *Latency of displaying characters typed into Emacs.*

between 10 and 15 minutes into the test, two big and one small Xine are active. As the small one needs less CPU, it is unperturbed under Linux, and only the big ones suffer. With the HuC scheduler, the big ones achieve a somewhat higher frame rate, at the expense of the small one that is also brought down to this level. Also note that new processes manage to get up to speed very quickly and don't fall behind those that are already running (the vertical lines indicate a Xine instance start or stop). This is especially noteworthy given that the processes are spawned by a script, and therefore do not start out identified as HuC.

9.3 Keeping Latency Low

The HuC scheduler not only allocates CPU time preferentially to HuC processes, it also does so promptly. The left of Fig. 10 shows the dispatch latency of various process types under loaded conditions when served by the Linux scheduler (we define dispatch latency as the period from entering the ready queue to being dispatched). The right side shows the results of running the same experiment with the HuC scheduler. The dispatch latencies of HuC-processes remains very low (typically < 2ms), regardless of the background load.

While the worst results are obtained for Emacs, they are still extremely good in absolute terms, and significantly lower than the 150ms threshold of human perception [Dabrowski and Munson 2001]. To verify this, Fig. 11 shows the total latency from the time a keystroke (as timestamped in the device driver) to when the corresponding character is displayed on the screen (as timestamped upon completion by the X server). The average grows to about 6ms, which corresponds to three dispatch latencies (from whatever process is running to X, then to Emacs, and back to X again). Before each keystroke we have verified that Emacs has timed out as a HuC process, and returned to the OTHER class. As such, the maximal measurements of up to 30ms include the time needed for the scheduler to identify Emacs as a HuC process again. But even this relatively high value is actually extremely good, and

$recent$	time to retain I/O status or count (8 sec)
$count_{out}$	metric for output (normalized pixels)
T_{update}	period of updates from X to kernel (1 sec)
$epoch$	maximum scheduling cycle (200 ms)
α	factor by which initial allocation grows (2)
$init$	initial allocation for new HuC (1% of epoch)
min_alloc	minimal allocation to OTHER process (1 tick)

Table III. *Configurable parameters of user-I/O monitoring and HuC scheduling.*

there is no real need for additional prioritization, which might come at the expense of other competing applications.

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.

9.4 Costs

One cost of running a scheduler is the direct overheads involved in its operations. We measured the following average values on a quiet system: calculating all priorities each second when new information arrives takes 69,276 cycles on average, starting a new epoch takes 14,878 cycles, the dispatch overhead is about 1,559 cycles, and moving a process between classes takes 1,425 cycles. The biggest overhead is obviously recalculating the priorities, but even this only totals to $\sim$ 0.01% overhead of the CPU's cycles on our 664MHz processor. As dispatch and epoch starts occur more often, their total effect is higher, and stands at 0.6% and 0.04% respectively. The total overhead of the scheduler activity is thus about 0.65% These results show little if any dependence on queue length with 0 to 10 stressors and several Xines running.

Another cost of the HuC scheduler is the possible effect on non-HuC processes. For example, such an effect occurs when network activity occurs in the background. Under sufficiently high loads, allocating the CPU preferentially to HuC activities may deprive the networking process from timely access to the CPU, reducing the achievable communication bandwidth. For example, we ran a test of Xine showing a 50 frames per second movie at double size together with a communicating process. The achieved Xine frame rate grows from 21 under linux to 37 with the HuC scheduler. As a result, the achieved communication bandwidth drops from 10.9MB/s to 3.9MB/s.

Finally, it should be noted that the HuC scheduler may be susceptible to some user counter-measures. For example, it is possible to envision an application that opens a small window for a short time just in order to perform some spurious output and gain priority. While we do not currently address such concerns, we note that most schedulers are actually open to such manipulations.

10. 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 [Etsion et al. 2004]. A possible alternative is 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.

Our main contribution is the proposal of a new metric to quantify user interest in running processes, based on their input and output events. This was then used to prioritize the processes and allocate CPU resources. Implementing this idea involved considerable work and modifications to the Linux system and X server, because it runs contrary to current designs. As such, it should be understood that many issues were left open. For example, the system has various parameters (some of which are listed in Table III) that need to be optimized. In particular, the choices we made were geared towards multimedia applications such as movie viewers. Other choices may be more appropriate for other workloads.

The idea of prioritization by user I/O opens many new and intriguing directions. For example, 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. It is also possible to consider a tighter coupling of the machine and the user. In our work, we need to infer what the user wants from input and output events. But one can also use devices that can provide even better measurements of user comfort or frustration, e.g. galvanic skin response meters and respiration sensors [Affective Computing Research Group at the MIT Media Lab], a webcam joint with a face recognition software to track if the user is looking at the screen [Dalton and Ellis 2003], or even pupil sensors to track which window the user is looking at, like some SLR cameras use for accurate focusing [Canon Inc.]. This will enable the user's mood and actions to directly affect system behavior.

REFERENCES

AFFECTIVE COMPUTING RESEARCH GROUP AT THE MIT MEDIA LAB. Research on Sensing Human Affect. URL: http://affect.media.mit.edu/AC_research/sensing.html.

BANACHOWSKI, S. A. AND BRANDT, S. A. 2002. The BEST Scheduler for Integrated Processing of Best-Effort and Soft Real-Time Processes. In *Multimedia Computing and Networking (MMCN)*.

BOVET, D. P. AND CESATI, M. 2001. *Understanding the Linux Kernel.* O'Reilly & Associates.

BRUNO, J., GABBER, E., ÖZDEN, B., AND SILBERSCHATZ, A. 1998. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In *Usenix Annual Technical Conf.* 235–246.

CANDEA, G. AND JONES, M. B. 1998. Vassal: Loadable scheduler support for multi-policy scheduling. In *Second USENIX Windows NT Symp.* USENIX, Seattle, WA, 157–166.

CANON INC. EOS ELAN 7N/7NE Camera. www.canon.com.

CHILDS, S. AND INGRAM, D. 2001. The Linux-SRT Integrated Multimedia Operating System: Bringing QoS to the Desktop. In *IEEE Real-time Technology & Apps. Symp.* 135.

CORBET, J., RUBINI, A., AND KROAH-HARTMAN, G. 2005. *Linux Device Drivers*, 3rd ed. O'Reilly & Associates.

DABROWSKI, J. R. AND MUNSON, E. V. 2001. Is 100 Milliseconds Too Fast? In *Conf. Human Factors in Computing Syst.* 317–318.

DALTON, A. B. AND ELLIS, C. S. 2003. Sensing User Intention and Context for Energy Management. In *Workshop on Hot Topics in Operating Systems*.

DUDA, K. J. AND CHERITON, D. R. 1999. Borrowed Virtual Time (BVT) Scheduling: Supporting Latency Sensitive Threads in a General Purpose Scheduler. In *Symp. Operating Systems Principles*. 261–276.

ETSION, Y., TSAFRIR, D., AND FEITELSON, D. G. 2003. Effects of Clock Resolution on the Scheduling of Interactive and Soft Real-Time Processes. In *Intl. Conf. on Measurement & Modeling of Computer Systems (SIGMETRICS)*. 172–183.

ETSION, Y., TSAFRIR, D., AND FEITELSON, D. G. 2004. Desktop Scheduling: How Can We Know What the User Wants? In *Intl. Workshop on Network & Operating Systems Support for Digital Audio & Video (NOSSDAV)*. 110–115.

EVANS, S., CLARKE, K., SINGLETON, D., AND SMAALDERS, B. 1993. Optimizing Unix Resource Scheduling for User Interaction. In *USENIX (Summer)*.

FEITELSON, D. G. AND NAAMAN, M. 1999. Self-Tuning Systems. *IEEE Software 16,* 2 (Mar/Apr), 52–60.

FLAUTNER, K., UHLIG, R., REINHARDT, S., AND UDGE, T. M. 2000. Thread-Level Parallelism and Interactive Performance of Desktop Applications. In *Arch. Support for Programming Languages & Operating Systems*. 129–138.

GALLMEISTER, B. O. 1995. *Posix. 4: Programming for the Real World*. O'Reilly & Associates.

GOEL, A., ABENI, L., KRASIC, C., SNOW, J., AND HAN WALPOLE, J. 2002. Supporting Time-Sensitive Applications on a Commodity OS. In *Symp. on Operating Systems Design & Impl.* 165–180.

GOYAL, P., GUO, X., AND VIN, H. M. 1996. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Symp. on Operating Systems Design & Impl.* 107–121.

GRAY, K. 2003. *Microsoft DirectX 9 Programmable Graphics Pipeline*. Microsoft Press.

MASSALIN, H. AND PU, C. 1990. Fine-Grain Adaptive Scheduling using Feedback. *Computing Systems*.

MAURO, J. AND MCDOUGALL, R. 2001. *Solaris Internals: Core Kernel Architecture*. Prentice Hall.

MCKUSICK, M. K., BOSTIC, K., KARELS, M. J., AND QUARTERMAN, J. S. 1997. *The Design and Implementation of the 4.4 BSD Operating System*. Addison Wesley.

MOSBERGER, D. AND PETERSON, L. L. 1996. Making Paths Explicit in the Scout Operating System. In *Symp. on Operating Systems Design & Impl.* 153–167.

NIEH, J., HANKO, J. G., NORTHCUTT, J. D., AND WALL, G. A. 1993. SVR4 UNIX Scheduler Unacceptable for Multimedia applications. In *Intl. Workshop on Network & Operating Systems Support for Digital Audio & Video (NOSSDAV)*.

NIEH, J. AND LAM, M. S. 1997. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *ACM Symp. on Operating Systems Principles*.

NIEH, J., VAILL, C., AND ZHONG, H. 2001. Vitrual-Time Round-Robin: An O(1) Proportional Share Schedulers. In *Usenix Annual Technical Conf.*

PAUL, B. 2000. Introduction to the Direct Rendering Infrastructure. http://dri.sourceforge.net/doc/DRIintro.html.

RAU, M. A. AND SMIRNI, E. 1999. Adaptive CPU Scheduling Policies for Mixed Multimedia and Best-Effort Workloads. In *Modeling, Anal. & Simulation of Comput. & Telecomm. Systems*. 252–261.

SHNEIDERMAN, B. 1998. *Designing the User Interface*, 3rd ed. Addison Wesley.

SILBERSCHATZ, A., GALVIN, P. B., AND GAGNE, G. 2004. *Operating System Concepts*, 7th ed. Addison Wesley.

SOLOMON, D. A. AND RUSSINOVICH, M. E. 2000. *Inside Windows 2000*, 3rd ed. Microsoft Press.

STEERE, D. C., GOEL, A., GRUENBERG, J., MCNAMEE, D., PU, C., AND WALPOLE, J. 1999. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *Symp. on Operating Systems Design & Impl.* 145–158.

TORVALDS, L., COX, A., AND MOLNAR, I. 2003. *Improving Interactivity*. http://kerneltrap.org/node/view/603. Linux Kernal Mailing List, Summarized Thread.

WALDSPURGER, C. A. AND WEIHL, W. E. 1994. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Symp. on Operating Systems Design & Impl.*

X CONSORTIUM. X Windows System. www.X.org.

ZHANG, Y. AND SIVASUBRAMANIAM, A. 2001. Scheduling Best-Effort and Real-Time Pipelined Applications on Time-Shared Clusters. In *ACM Symp. on Parallel Algorithms and Architectures(SPAA)*. 209–218.