Window-Constrained Process Scheduling for Linux Systems

Richard West
Computer Science Department
Boston University
Boston, MA 02215
richwest@cs.bu.edu

Ivan Ganev and **Karsten Schwan**
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{ganev,schwan}@cc.gatech.edu

Abstract

This paper describes our experience using Dynamic Window-Constrained Scheduling (DWCS) [13, 14, 12, 11] to schedule processes (and threads) on available CPUs in a Linux system. We describe the implementation of a kernel-loadable module that replaces the default Linux scheduler. Each process scheduled using DWCS has a request period of T time units and a 'window-constraint', x/y. The end of one request period and the start of the next denotes a deadline by which time a process must be serviced for one quantum, of K time units. Under these constraints, DWCS attempts to guarantee that each process receives at least $(y-x)K$ units of CPU time every window of yT time units. That is, DWCS tries to guarantee that each process misses no more than x deadlines every window of y requests for service, and each time a process is serviced it receives K units of CPU time.

DWCS can produce a feasible schedule under certain constraints, when the least upper bound on resource utilization [8] does not exceed 100% *and* each process is serviced in fixed size time slots [12]. We show that DWCS is capable of successfully scheduling CPU- and I/O-bound processes in Linux more than 99% of the time, when a feasible schedule is theoretically possible. Unlike in the theoretical case, interrupt handling, context-switching, scheduling latency and unpredictable management of other resources besides the CPU affect the predictable scheduling of processes. We discuss several approaches that we are considering, to account for system overheads and provide predictable real-time scheduling to processes in Linux.

1 Introduction

For many real-time applications, it is important that a process (or thread) receives a guaranteed share of the CPU over a finite window of time. Rather than guaranteeing a process receives all its allocation of the CPU in one instance, it is often important to overlap the execution of multiple processes so that each receives a "fair share" of the CPU over fixed time intervals. This ensures that processes make progress at guaranteed rates. Consequently, it is important for processes to be executed in time slices over predictable time intervals. However, it is typically not practical in scalable systems, to ensure that *every* process time slice is executed at predictable times. For such systems, it is possible to allow a finite number of process time slices to be executed later than desired as long as: (a) most time slices are executed at the desired rate and, (b) the entire process receives a guaranteed share of the CPU over a finite window of time. For example, a video server might support thousands of client requests, and the threads serving these requests need to compress and transmit video frames at guaranteed rates in order to meet client-level service constraints. If some frames are generated later than desired and are consequently discarded, the client might detect some dropouts in the playback of the video stream. This might be acceptable as long as the number of consecutive losses of video frames does not exceed a client-specified threshold. If the threshold is exceeded it may be impossible to interpret the received video sequence at the client. Likewise, if frames are generated too fast, playout buffers at the client might overflow and some frames will again be lost.

We have developed an algorithm called Dynamic Window-Constrained Scheduling (DWCS) [13, 14, 12] to support real-time and multimedia applications requiring rate-based service constraints like those described above. DWCS was originally designed as a packet scheduler to provide (m,k)-*firm* deadline guarantees [5] and fair queueing [2, 15, 3, 1, 4, 9, 10], for loss and delay constrained traffic streams. This was particularly useful for servicing multimedia audio and video streams, which can tolerate a certain fraction of lost information, as long as consecutive losses are limited. DWCS was then extended to guarantee (m,k)-*hard* deadlines (or, equivalently, hard guarantees that no more than x missed packet deadlines occur for every window of y consecutive packets in a given stream). This is similar to the Rate-Based Execution (RBE) model [6], but in that work there is no notion of missing, or discarding service requests. More recently, DWCS has been applied as a process (or thread) scheduler in the Linux kernel [11].

For process scheduling, consider that each process has service constraints in terms of a request period, T, a 'window-constraint', x/y, and a service quantum, C. The end of one request period and the start of the next denotes a deadline by which time a process must be serviced for one quantum (or time slice), of C time units. Under these constraints, DWCS attempts to guarantee that each process receives at least $(y-x)C$ units of CPU time every non-overlapping (i.e., adjacent) window of yT time units. That is, DWCS tries to produce a *feasible* schedule in which each process misses no more than x deadlines every window of y requests for service, and each time a process is serviced it receives C units of CPU time. This means that DWCS can service processes that require execution multiple times (such as periodic processes), as well as processes that only execute once. In either case, a process is serviced at the granularity of a quantum of C time units. Moreover, a process must be serviced for one quantum in a single request period otherwise a deadline is missed.

As a packet scheduler, DWCS attempts to guarantee that no more than x packets are serviced late, for every y consecutive packets in the same stream requiring service. A late packet is considered lost since it is useless to a recipient. That said, there are many similarities between process and packet scheduling and DWCS has important properties pertaining to both. Consider that a process or packet stream, P_i, has service constraints T_i, x_i/y_i and C_i. DWCS can guarantee a *feasible* schedule for all P_i, where $1 \leq i \leq n$, if the minimum utilization factor, U, equals $\sum_{i=1}^{n} \frac{(1-x_i/y_i)C_i}{T_i}$ and $U \leq 1.0$. We impose a restriction that C_i is less than or equal to K time units, where K represents the maximum-sized service quantum. For theoretical guarantees, we assume that at most one packet or process time slice is serviced every K time units. Moreover, the scheduler must be invoked at least once every K time units.

To guarantee a feasible schedule, $U \leq 1.0$, $C_i \leq K$, and $T_i = q_i K$ for all positive integers q_i, where $1 \leq i \leq n$. If these conditions hold, a set of processes or streams, $P = \{P_1, \cdots, P_n\}$, can be feasibly scheduled with service constraints $(T_1, x_1/y_1, C_1) \cdots (T_n, , x_n/y_n, C_n)$, respectively. However, these constraints may have to be translated into their *canonical* form. That is, for a process or stream P_i with service constraints $(T_i, x_i/y_i, C_i)$, where $T_i = q_i K$, the canonical set of constraints are $(T_i^*, x_i^*/y_i^*, C_i^*)$, with $T_i^* = K$, $x_i^* = y_i(q_i - 1) + x_i$, $y_i^* = q_i y_i$ and $C_i^* = C_i$.

The generation of these equivalent constraints can be done internally by the scheduler, although at present this is not done so it is possible that theoretically feasible schedules are not always obtainable in practice. Moreover, system overheads affect these theoretical bounds. In any case, it is theoretically possible to guarantee a minimum CPU utilization of $(1 - x_i/y_i)C_i/T_i$ or, equivalently, $(1 - x_i^*/y_i^*)C_i^*/T_i^*$ to P_i. Throughout the remainder of the text we will assume $C_i = C_i^* = K$ for all i, but in practice it is acceptable for C_i (and the equivalent C_i^*) to be less than K at the cost of potentially reduced resource utilization. It should also be pointed out that the notion of generating canonical service constraints, to guarantee feasible schedules where possible, is not discussed in the related paper [12] but is included here for completeness.

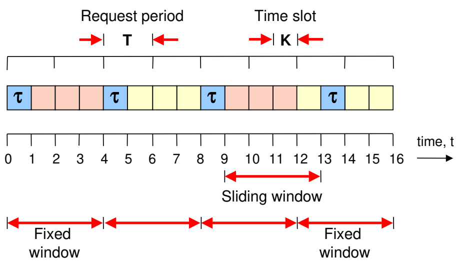

FIGURE 1: *An example schedule for a process.*

Figure 1 shows an example of a process that is successfully scheduled to meet its service constraints, where $T = 2$, $x/y = 1/2$ and the time slice is $C = K = 1$ time units. The scheduler executes once every time slot, where one time slot is K time units. All request periods are multiples of a time slot. Observe that the process is successfully scheduled according to its service constraints, since it receives one unit of service time every fixed window of four time units. For the current version of DWCS, sliding windows are not supported. In this example, there is a window of four time units between time $t = 9$ and $t = 13$ where the process is not serviced. In the worst-case, DWCS services a process at the beginning of one fixed window and the end of the next fixed window. This means that for a process, P_i, there can never be a sliding window larger than $2(T_i - C_i)$ where P_i is not serviced. For applications which require bounded delay variation, or jitter, this is an important property. In the future, we hope to extend DWCS to support sliding windows.

Contributions: This paper describes our experience using DWCS to schedule processes (and threads) on available CPUs in a Linux system. We describe the implementation of a kernel-loadable module that replaces the default Linux scheduler. We have taken the approach of modifying an off-the-shelf version of Linux rather than using a custom real-time system, or an approach such as RTLinux [7], to measure the delay guarantees that are possible by simply changing the kernel scheduler. Using DWCS, processes are scheduled to meet their service constraints over finite windows of time. By contrast, the default kernel scheduler does not consider explicit time-constraints on the execution of processes.

We show that DWCS is capable of successfully scheduling CPU- and I/O-bound processes in Linux more than 99% of the time, when a feasible schedule is theoretically possible. Unlike in the theoretical case, interrupt handling, context-switching, scheduling latency and unpredictable management of other resources besides the CPU affect the predictable scheduling of processes. We discuss several approaches that we are considering, to account for system overheads and provide predictable real-time scheduling to processes in Linux.

The next section describes process scheduling in Linux using DWCS. A description of the implementation of DWCS in Linux is included. Section 3 describes some of the experiments we have conducted and the results we have obtained so far. Some of the issues needed to be addressed to guarantee predictable real-time scheduling in Linux are discussed in Section 4. Finally, conclusions and future work are described in Section 5.

2 Process Scheduling Using DWCS

We have implemented an (m, k)-*hard* version of DWCS for Linux [12]. The Linux DWCS scheduler compares the service constraints of pairs of processes (or, equivalently, threads) and selects the next process *with the earliest deadline* for execution. The current deadline of a process refers to the time by which the next time slice of that process must complete execution. The deadline d_i, of a process P_i, is derived from the current time t, and the request period T_i. That is, $d_i = t + T_i$, where t is a multiple of the slot time K. In other words, t represents the time when a scheduling decision is made and all decisions must be made on slot boundaries.

For two or more processes with the earliest deadline, the process with the lowest *current* window-constraint is chosen. Intuitively, this means that a process can tolerate fewer deadline misses over a given period of time. If there is a tie between deadlines and window-constraints, the process P_i with the lowest window-numerator, x_i is selected. Essentially, this means that one process can tolerate the same fraction of missed deadlines as some other process but over a smaller window of deadlines. The window-constraints of processes are adjusted over time, so it may be possible that two or more processes have zero-valued window constraints and the same deadlines at some time instant. In this case, the process with the highest window-denominator, y_i, is chosen for execution, since it can tolerate no missed deadlines over a larger window of consecutive deadlines. All other cases are scheduled first-come-first-serve.

For each process P_i, serviced for a time slice before its deadline (i.e., a process that is serviced in its current request period), the window-constraint x_i/y_i is adjusted dynamically to, in effect, reduce the urgency of servicing the same process again when another more "critical" process requires service.

Let x_i be P_i's *original* window-numerator, while y_i is its original window-denominator. These are the values first set for a DWCS process. Let x'_i and y'_i denote the *current* window-numerator and current window-denominator as the process is executed. Before P_i is serviced, $x_i = x'_i$ and $y_i = y'_i$. Then, if P_i is serviced before its deadline, the current window-constraint, x'_i/y'_i, is adjusted as follows:

Rule (A) – Window-constraint adjustment when P_i is serviced in its current request period:

if $(y'_i > x'_i)$ then $y'_i = y'_i - 1$;
else if $(y'_i == x'_i)$ and $(x'_i > 0)$ then
    $x'_i = x'_i - 1; y'_i = y'_i - 1$;
if $(x'_i == y'_i == 0)$ or (P_i is tagged) then
    $x'_i = x_i; y'_i = y_i$;
if (P_i is tagged) then reset tag;

At this point in time the current window-constraint, x'_j/y'_j, of any other process, $P_j \mid j{\neq}i$, having missed its deadline, is adjusted as follows:

Rule (B) – Window-constraint adjustment when $P_j \mid j \neq i$ is not serviced in its current request period:

if $(x'_j > 0)$ then
    $x'_j = x'_j - 1; y'_j = y'_j - 1$;
    if $(x'_j == y'_j == 0)$ then $x'_j = x_j; y'_j = y_j$;
else if $(x'_j == 0)$ and $(y_j > 0)$ then
    $y'_j = y'_j + \epsilon$;
    Tag P_j with a violation;

If a process violates its original window-constraint, it is tagged for when it is next serviced. A tagged process has a `DWCS_VIOLATION` flag set. This ensures that the process is never starved of service even in overload. In fact, we have shown in prior work the worst-case delay bound for overload situations [12].

The value of ϵ is 1 in the current Linux DWCS implementation. In general, this value can be set to higher positive values to increase the urgency of servicing a process that violates its window-constraints. Window-constraint violations occur during overload, but it is also possible for violations to occur in other situations. The include situations when service constraints are not translated into their canonical form, or when there is a mix of *static* and *dynamic* priority processes. Observe that processes with finite deadlines are dynamic priority processes, while those with 'infinite' deadlines (i.e., they have request periods of -1) are static priority processes. A static priority process is *non-time-constrained* and its window-constraint is used as the priority: the lower the window-constraint the higher the priority. If we take the approach that static priority processes are only serviced when no time-constrained processes need servicing in their current request periods, it is possible to produce feasible schedules when $U \leq 1.0$ [12].

As well as supporting a mixture of static and dynamic priority processes, DWCS also supports pure earliest-deadline-first (EDF) scheduling. In this mode all processes have original window-constraints equal to 0/0 and finite deadlines (i.e, positive-valued request periods). This means that each and every P_i has a corresponding $x_i = 0$ and $y_i = 0$ for the scheduler to operate in EDF mode.

Although DWCS can support different scheduling modes, it is primarily intended to support window-constrained processes. To provide window-constrained guarantees, each and every process, P_i, must have an original window-constraint, x_i/y_i that is not 0/0 , and a finite deadline derived from a positive-valued request period, T_i. These constraints may then need to be translated into their canonical form to produce a feasible schedule. We can now show how DWCS works, using the following pseudo-code:

```
Let Pi = process i
    di = current deadline of Pi
    Ti = request period of Pi
    Wi'= current window-constraint of Pi

while (TRUE) {
  for (each ready process)
      find process, Pi, with the earliest
      deadline; if two or more processes
      have the same deadline, resolve ties
      by comparing window-constraints, as
      described earlier;

  service Pi for one time slice;
  adjust Wi' according to Rule (A);
  /* Adjust deadline of next time slice     */
  /*   or instance of Pi.                   */
  di = di + Ti;
  for (each process Pj, other than Pi,
       missing its deadline) {
    while (deadline missed) {
      adjust Wj' according to Rule (B);
      /* Adjust deadline of next time slice */
      /* or instance of Pj.                 */
      dj = dj + Tj;
    }
  }
}
```

A process is ready when the current time is greater than or equal to the start time of the current request period for the process. All request periods begin on scheduling point boundaries. Once a process has been serviced in the current request period, it is not ready for further execution until the start of the next request period. This forces the scheduler to operate in *non-work-conserving mode*, thereby delaying the execution of a process even if there are no other processes to execute. The non-work-conserving mode is necessary to (a) guarantee that a process is not granted more than its required share of CPU time at the cost of other processes and, (b) guarantee tighter bounds on the delay variation between servicing consecutive process time slices. However, to allow some processes to continue when there are no other processes to execute, Linux DWCS supports a work-conserving mode of operation. If the `DWCS_WORK_CONS` flag is set, then potentially multiple time slices of a given process can execute in a single request period. This will only happen if all other time-constrained processes have been serviced in their current request periods.

2.1 Linux DWCS Implementation

DWCS is currently implemented as a kernel-loadable module in Linux. We chose this approach to be able to easily modify the algorithm without continually recompiling the kernel. However, some modifications were necessary to the core kernel, including exporting additional symbols for module linkage via `kernel/ksyms.c`. Due to the kernel modifications, we have so far only implemented DWCS for kernel versions 2.2.7 and 2.2.13. Future releases will no doubt work with the latest kernels. A patch file for the appropriate kernel version is available from the DWCS website [11].

To redirect the `schedule()` routine in `kernel/sched.c` to invoke the DWCS scheduler, a flag called `DWCS_module_loaded` is set. This flag is set by first loading the DWCS module (`dwcs.o`) and then invoking a new system call, `load_scheduler()` from within a program. A command-line executable (`load_scheduler`) is included with the distribution available from the website that activates the DWCS scheduler using this system call. A corresponding `unload_scheduler()` system call deactivates DWCS and reverts back to the default scheduler. In the future, we are considering using `ioctl` calls to replace the system calls that we have added to the kernel. A third and final 'dummy' system call (`DWCS_scheduler()`) is used to redirect control to the DWCS scheduler when activated. This is not a system call that should be used by user-level programs but is instead used as a way to redirect control to the module code for the `DWCS_scheduler()` function. It is a function that is registered in the system call table and declared as a dummy function in the core kernel (see `kernel/sys.c` after applying the DWCS patch). Upon loading the module, the `init_module()` redirects the function pointers for the new system call entries in the `syscall` table to the actual functions in the module. This procedure allows functionality to be added to a running kernel while ensuring the core kernel can be re-linked without experiencing unresolved symbols. Although this is not really what modules are designed for, it is a useful way to extend the behavior of the kernel and is a property we are exploiting to build an extensible real-time system known as 'Dionisys'.

A /proc/dwcs file entry can be established by selecting `CONFIG_PROC_DWCS` when configuring a DWCS-patched kernel. This provides access to various status information, including service constraints, deadlines missed and window-constraint violations for processes currently in the run queue. The `sched_setscheduler()` function can be used to set scheduling parameters for processes once the DWCS scheduler is activated. The scheduling policy should be set to `SCHED_DWCS` and a pointer to a `struct sched_param` structure, passed as an argument to `sched_setscheduler()`, should include the process's service constraints. The revised `struct sched_param` structure, in `linux/sched.h` is as follows:

```
struct sched_param {
  int sched_priority;   /* Original member.   */
  unsigned long period; /* Request period.    */
  unsigned int own;     /* Window numerator.  */
  unsigned int owd;     /* Window denominator.*/
  unsigned int flags;   /* {See below}        */
  unsigned long service_time; /* Time quantum.*/

};
```

The default values of these constraints are assigned to every process, regardless of whether or not DWCS is active or even resident in the kernel. Changes to `kernel/fork.c` establish default service constraints for newly-created processes (and threads). These values are stored in the process descriptor, so modifications to the `struct task_struct` structure in `linux/sched.h` were required. These values ensure all processes are initially non-time-constrained and work-conservative (i.e., the `DWCS_WORK_CONS` flag is set). By overriding these values using `sched_setscheduler()` it is possible to make a process time-constrained, by assigning it a positive request period. The window-constraints, service quantum and flags can be changed accordingly. To date, the `flags` member of `struct sched_param` can be 0 or a logical OR combination of `DWCS_WORK_CONS` and `DWCS_NON_PREEMPTIVE`. The latter flag is used for situations where a process's service quantum is greater than the scheduling granularity. If it is required that the scheduler does not preempt a process (in favor of another process) until its time slice has expired then `DWCS_NON_PREEMPTIVE` must be set. This is similar to the behavior of the default scheduler.

The majority of the scheduler code is in `kernel/dwcs.c`. Upon each invocation, the scheduler compares all processes in the run queue according to the DWCS selection rules, and updates service constraints as necessary. The current implementation queues processes on the default run queue, which is a doubly-linked list. For future versions of Linux DWCS, we are considering using heaps [14] for queueing real-time processes. Moreover, by using two run queues: one for real-time processes and another for all other processes, we can reduce the scheduling latency for time-critical processes. Only when there are no real-time processes waiting in the corresponding run queue can we select other processes.

3 Experimental Evaluation

We ran a series of experiments on one CPU of a $400Mhz$ Pentium II (Deschutes) machine, with 512KB L2 cache, 1GB PC100 SDRAM, one Adaptec AIC-7860 Ultra SCSI host adapter and one SEAGATE ST39102LC (8GB) hard drive. The machine was configured with Linux 2.2.13. Different numbers of I/O- and CPU-bound processes were scheduled for execution, each having different service constraints. For each experimental run, we recorded performance information including the number of deadlines missed and window-constraint violations.

To ensure that all real-time processes were synchronized, we took the following procedure: a parent process forked the desired number of child processes, then collected initial statistics from `/proc/stat`, and finally loaded a special kernel module to start the experimental run. The start module's `init_module()` function assigned appropriate service constraints to each child process, overriding the default values. Then a set of signals were sent from the module to all child processes, which at this time were waiting on a barrier. The start module terminated and the child processes were able to proceed with their CPU- and/or I/O-bound operations. The parent process waited for the completion of all child processes and finally logged the `/proc/stat` and `/proc/dwcs` files. For the experimental runs, `/proc/dwcs` was modified to provide more detailed output to a main memory buffer.

Each I/O-bound process read 1000 raw bitmaps from disk, while each CPU-bound process calculated an FFT on a matrix of 4 million floating point numbers. Each process averaged about 60 seconds to execute on a quiescent system. In each experiment, the utilization, U, was calculated by varying the values of T_i (the request period), x_i/y_i (the window-constraint), and the number of processes, n, under the assumption that each process $P_i \mid 1 \leq i \leq n$ received a unit quantum of service, K, equal to one jiffy (i.e., one clock tick of about $10mS$) every time it was executed [1]. That is, $U = \sum_{i=1}^{n} \frac{(1-x_i/y_i)K}{T_i}$, where parameters T_i, x_i/y_i and n were varied over the ranges $[2K, \cdots, 64K]$, $[1/2, \cdots, 1/10]$, and $[2, \cdots, 64]$, respectively. Observe that U represents the *minimum* utilization needed to guarantee a feasible schedule. However, the actual demand for CPU cycles, assuming no missed deadlines is $U_{max} = \sum_{i=1}^{n} \frac{K}{T_i}$, which can be significantly greater than 1.0.

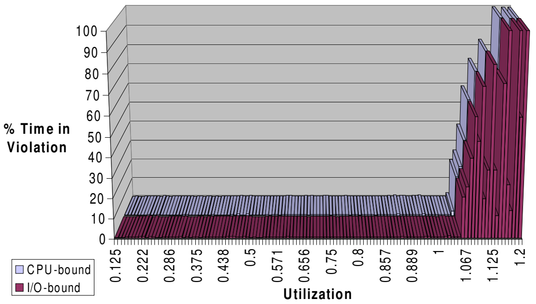

FIGURE 2: *Average percentage of time each process spends violating its service constraints.*

Figure 2 shows the percentage of time a series of I/O- and CPU-bound tasks were in violation of their service constraints, when utilizations were varied between 0.0 and 1.2. The x-axis shows a series of sample utilizations constructed from various combinations of T_i, x_i/y_i and n and is not a linear scale. For these experiments, DWCS successfully schedules CPU- and I/O-bound processes in Linux more than 99% of the time, when $U \leq 1.0$. That is, window-constraint violations occur less than 1% of the total execution time of all processes, when it is theoretically possible to guarantee no violations. It should be noted that a feasible schedule is theoretically possible when $U \leq 1.0$, assuming that scheduling and other system overheads are negligible.

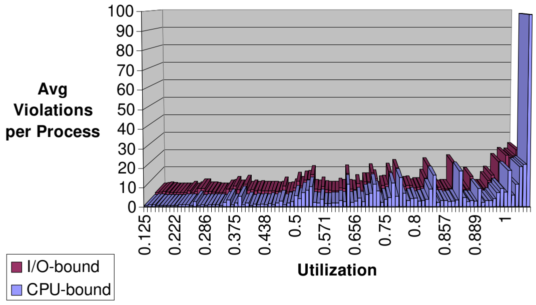

FIGURE 3: *The average violations per process, for different utilizations using DWCS as the Linux scheduler when the system is otherwise quiescent.*

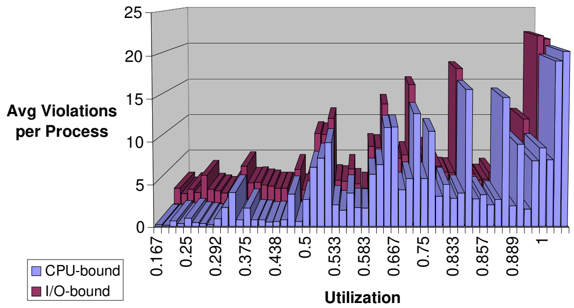

FIGURE 4: *The average violations per process, for different utilizations using DWCS as the Linux scheduler when the system is flood-pinged.*

In Figure 3, no more than 20 violations ever occur for a single process, on average (calculated over 30 runs of each sample utilization), until the utilization reaches 1.0. In all cases, CPU- and I/O-bound performances are similar. These results are for the case when the system is otherwise in a quiescent state, and represent the actual number of violations for the experiments shown in Figure 2. Note that Figure 3 only shows results for utilizations as they just reach 1.0. All sample values at or after 1.0 on the x-axis represent utilizations of 1.0 derived from varying the

[1] Although a service quantum of one jiffy seems small, we wanted to see just how well Linux could perform when the system overheads were a significant fraction of the service quantum. Having a small quantum makes overheads such as context-switching, scheduling latency and interrupt handling more significant.

request periods, window-constraints and number of processes. The CPU-bound processes suddenly incur a large number of violations when the utilization is 1.0 and the number of processes reaches 64. This is due to the greater demand for CPU cycles by these processes, whereas the I/O-bound processes tend to block before using their entire time slice of $K = 1$ jiffy. Consequently, I/O-bound processes don't show the sudden increase in violations around the overload point.

By contrast, Figure 4 shows the number of window-constraint violations when the host scheduling the CPU- and I/O-bound processes is flood-pinged by a remote host. This causes thousands of interrupts to be generated during the execution of each process, thereby causing significant system overheads. However, the number of violations still remains below 20 for utilizations below 1.0.

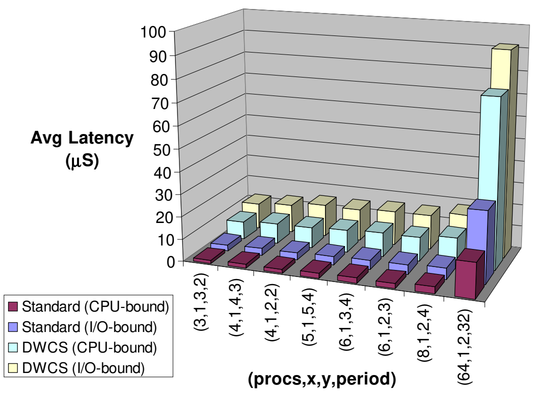

FIGURE 5: *Average latency of the standard Linux scheduler and DWCS for CPU- and I/O-bound processes under varying utilizations.*

Finally, Figure 5 shows the scheduling latency for DWCS versus the standard Linux scheduler under varying utilizations. Both DWCS and the standard scheduler incur linear time penalties due to the run queue implementation. However, DWCS is not optimized and there is more state manipulation required, to update service constraints for processes on a regular basis. From past experience [14], we believe that by implementing a separate (possibly heap-based) run queue for real-time processes we can significantly reduce the scheduling overheads when using DWCS. Moreover, DWCS has shown to be fairly insensitive to slight variability in the intervals between which the scheduler is invoked. This means that we can reduce the rate of invocation of the scheduler, and consequently its overhead, and still achieve minimal service violations. Naturally, there is a limit on the time we can delay the scheduler otherwise all service guarantees will be void.

4 Providing Better Service Guarantees

Although the results for the Linux implementation of DWCS look encouraging, there are still some service violations when it is theoretically possible to eliminate them. Even if system overheads are accounted for in the utilization calculations, there can still be violations. This is a direct consequence of several factors concerning the Linux operating system. Linux does not support fixed preemption points, so the scheduler is not guaranteed to execute after fixed intervals of time. Also, the resource management features of Linux are, in general, not predictable. That is, even if DWCS is used as a real-time scheduler, acquisition of resources such as memory, semaphores and locks can take an arbitrary amount of time. Moreover, paging activity can incur large delays that affect predictable scheduling, which is one reason why many real-time systems do not support virtual memory.

Rather than attempting to rewrite a major part of the kernel, we intend to compensate for the lack of predictable resource management. For example, we have tried to compensate for variability in the time between scheduler invocations. Observe that the standard scheduler is typically called when a process either terminates, blocks or completes its current service quantum [2]. The scheduler is activated by calling `schedule()` in `kernel/sched.c`, but this is only possible upon return from a kernel control path to user-level. In other words, the scheduler is activated only if the last saved context was executing in user-mode (see `ret_from_intr` in `kernel/entry.S`). All these conditions make it possible for the scheduler to be called multiple times per jiffy or just once every few jiffies. Such variability is problematic for real-time scheduling.

To deal with the variability in the time between scheduler invocations, a flag (`DWCS_reschedule`) is set every time `do_timer()` is called, which is once every clock tick, or jiffy. A check is then performed in `kernel/entry.S` to see if this flag is set and, if so, the scheduler is called:

```
ret_with_reschedule:
 movl SYMBOL_NAME(DWCS_reschedule),%eax
 cmpl $0,%eax
 jne reschedule

 ## The remaining code is unchanged ##
```

[2] The default quantum is about $210mS$.

```
    ...
## End of entry.S                          ##
```

Finally, the `DWCS_reschedule` flag is reset in `schedule()`.

The DWCS code also checks to see that the value of `jiffies` has been incremented before updating service constraints. This is to ensure that service constraints are not incorrectly updated if the scheduler is invoked multiple times in the same clock tick.

Our modifications make the calls to the scheduler more regular. Unfortunately, there still might be multiple clock ticks between calls to the scheduler, because (as stated above) the scheduler cannot be called from within nested kernel control paths. To compensate further, we are considering a way to adjust a process's service constraints by some function of the time between scheduler invocations. However, this is something we are still investigating at this time.

Observe that in the experiments described in the previous section all utilizations, U, were calculated under the assumption that a process quantum was K time units. In fact, we assumed it to be $K = 1$ jiffies but the actual value of K depends on the rate at which the scheduler runs. Due to the variability of scheduler invocations, one process might receive more than K units of CPU time at the cost of other processes that suffer delayed execution. This means that even though two or more processes have the same service constraints, they may actually experience different CPU utilizations over finite windows of time. Consequently, two processes with the same service constraints might complete at significantly different times even though they are initially ready to execute at the same time. Although no results are shown in this paper, we observed that the CPU-bound processes are more prone to large variations in their completion times, even when they all have the same service constraints. This is primarily because they need a lot more CPU cycles than I/O-bound processes in order to complete their execution. Observe that the scheduler does not differentiate between time spent in interrupts and time spent executing processes, so a process's quantum can be consumed by interrupt processing. This affects the progress of CPU-bound processes more than I/O-bound processes, which may actually be blocked while interrupt processing is taking place. Interestingly enough, I/O-bound processes might block before completing their service quanta but this seems to have less effect on the overall schedule than variability in the scheduling points.

We are also considering the use of *logical time* for all scheduling decisions. This will ensure that the schedule order is the same as the theoretical order but the processes may actually start and end in real-time later than desired. We are currently looking at measuring the delays between scheduling points and compensating for these, while using logical time to make scheduling decisions. Since significant scheduling delays are due to processing interrupts (particularly those issued by I/O devices) we are considering modification to `do_IRQ()`, to account for the time spent in interrupts. By accounting for interrupt overheads in this way, we can determine how much progress a process makes in a given time slice. If a process has lost a lot of time in its current time slice to processing interrupts, we can continue its execution beyond the time at which its time slice would otherwise expire. When a process has exhausted its time slice by executing at user-level, we can then schedule another process. This will ensure that all processes make progress according to their service constraints.

5 Conclusions and Future Work

This paper describes our experience using DWCS to schedule processes (and threads) on available CPUs in a Linux system. We have shown how to implement a kernel-loadable module that replaces the default Linux scheduler. We have taken the approach of modifying an off-the-shelf version of Linux rather than using a custom real-time system, to measure the delay guarantees that are possible by simply changing the kernel scheduler. Using DWCS, processes are scheduled to meet their explicit delay and window constraints. By contrast, the default kernel scheduler does not consider explicit delay constraints on the execution of processes.

We have shown that DWCS is capable of successfully scheduling CPU- and I/O-bound processes in Linux more than 99% of the time, when a feasible schedule is theoretically possible. Unlike in the theoretical case, interrupt handling, context-switching, scheduling latency and unpredictable management of other resources besides the CPU affect the predictable scheduling of processes.

Several approaches to account for system overheads and provide predictable real-time scheduling have been discussed. One such approach attempts to guarantee that the scheduler is invoked at fixed points in time (e.g., once per jiffy, or every K time units). However, the scheduler cannot be invoked if there are nested kernel control paths, so it is still possible for multiple clock ticks to pass between calls to the scheduler. To compensate for this, we are considering ways to measure the delay between scheduler invocations, and adjust the service constraints for processes accordingly. Finally, we intend to measure the time spent in interrupts, to compensate for these overheads and guarantee progress of processes in proportion to their service constraints.

References

[1] J. C. Bennett and H. Zhang. WF^2Q: Worst-case fair weighted fair queueing. In *IEEE INFOCOMM'96*, pages 120–128. IEEE, March 1996.

[2] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair-queueing algorithm. *Journal of Internetworking Research and Experience*, pages 3–26, October 1990.

[3] S. Golestani. A self-clocked fair queueing scheme for broadband applications. In *INFOCOMM'94*, pages 636–646. IEEE, April 1994.

[4] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *IEEE SIGCOMM'96*. IEEE, 1996.

[5] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m,k)-firm deadlines. *IEEE Transactions on Computers*, April 1995.

[6] K. Jeffay and S. Goddard. A theory of rate-based execution. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS)*, December 1999.

[7] R.-T. Linux. http://www.rtlinux.org.

[8] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 1973.

[9] X. G. Pawan Goyal and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *2nd Symposium on Operating Systems Design and Implementation*, pages 107–121. USENIX, 1996.

[10] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Real-Time Systems Symposium*. IEEE, December 1996.

[11] R. West. Linux DWCS: http://www.cc.gatech.edu/~west/dwcs.html.

[12] R. West and C. Poellabauer. Analysis of a window-constrained scheduler for real-time and best-effort packet streams. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, December 2000.

[13] R. West and K. Schwan. Dynamic window-constrained scheduling for multimedia applications. In *6th International Conference on Multimedia Computing and Systems, ICMCS'99*. IEEE, June 1999. Also available as a Technical Report: GIT-CC-98-18, Georgia Institute of Technology.

[14] R. West, K. Schwan, and C. Poellabauer. Scalable scheduling support for loss and delay constrained media streams. In *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium*. IEEE, June 1999.

[15] H. Zhang and S. Keshav. Comparison of rate-based service disciplines. In *Proceedings of ACM SIGCOMM*, pages 113–121. ACM, August 1991.