Using Windows NT for Real-Time Applications: Experimental Observations and Recommendations*

Krithi Ramamritham, Chia Shen(†),
Oscar González, Subhabrata Sen and Shreedhar Shirgurkar
Computer Science Department
University of Massachusetts
Amherst, MA 01003 (krithi, ogonzale, sen, shree@cs.umass.edu)
(†) MERL - A Mitsubishi Electric Research Lab
Cambridge, MA 02139 (shen@merl.com)

Abstract

Windows NT was not designed as a real-time operating system, but market forces and the acceptance of NT in industrial applications have generated a need for achieving real-time functionality with NT. As its use for real-time applications proliferates, based on an experimental evaluation of NT, we quantitatively characterize the obstacles placed by NT. As a result of these observations, we provide a set of recommendations for users to consider while building real-time applications on NT. These are validated by the use of NT for a prototype application involving real-time control that includes multimedia information processing. The results of the above study should provide system designers with guidelines, as well as insight, into the design of an architecture based on NT for supporting applications with components having real-time constraints.

1. Introduction

Ideally, for supporting real-time applications, a real-time operating system ought to be used. However, market forces and the acceptance of NT in industrial applications have generated a need for achieving real-time functionality using NT. Many real-time systems and applications desire to use NT as is, so as not to incur the overhead of either the installation of other kernels and facilities beyond those provided in the standard NT package, or the usage of some other APIs that run in parallel with NT's Win32.

The purpose of this paper is to examine NT from the perspective of real-time constraints and systematically arrive at guidelines and recommendations that will be useful for real-time system designers as they build applications using NT. Because NT was not designed with predictability in mind, it is neither advisable nor feasible to use NT for hard real-time applications, for example, at the controller level with sub-millisecond precision. But, used judiciously, it may be useful for applications that (1) can tolerate occasional deadline misses, and (2) have delay/response time requirements in the tens to hundreds of milliseconds range such as those described in [5, 7, 9]. To this end, a key element of our study is to see to what extent the unpredictable parts of NT can be "masked".

We begin, in Section 2, by giving an overview of the capabilities of NT that are potentially useful to real-time system builders. We then critically evaluate their performance characteristics via a series of experiments. The experiments and the observations derived from them are summarized in Section 3. These observations are then used to build a prototype of an application involving real-time control that includes multimedia information processing. Section 4 describes the components of the prototype along with the resulting assessment of NT's suitability for such real-time applications. This section also contains a set of guidelines and recommendations that emerge from the experimental evaluation of NT for real-time uses.

In this work our intention is to evaluate NT "as is" for real-time uses. On the other hand, various commercial efforts are aggressively working towards extending NT (typically by modifying NT or its underlying layers [8], [14]). Section 5 critically evaluates these efforts as well as Windows CE (intended for embedded applications) and other research efforts.

Although there have been efforts in qualitatively analyzing Windows NT's suitability for real-time applications [11], this paper is the first effort in quantitatively studying NT via an experimental approach. We hope this paper, based on systematic experimental evaluations, will provide

*Supported in part by NSF under Grant CDA-9502639 and by MERL-A Mitsubishi Electric Research Laboratory.

guidelines, as well as insight, needed to design a software architecture based on Windows NT capable of supporting hard and soft real-time tasks. This architecture is the foundation of an operating systems test-bed that we are currently building to help in the development and evaluation of scheduling techniques that support, in a synergistic manner, the needs of tasks having a variety of real-time and non real-time requirements.

2. Windows NT as a Real-Time OS

NT's use of threads and thirty two possible priority levels can be helpful in constructing real-time applications. So we discuss these first.

Each process belongs to one of the following priority classes: IDLE, NORMAL, HIGH and REALTIME. By default, the priority class of a process is NORMAL. Processes that monitor the system, such as screen savers or applications that periodically update a display, use the priority class IDLE. A process with HIGH priority class has precedence over a process with NORMAL priority class. The REALTIME priority class is provided as a support for real-time applications.

Windows NT assigns a scheduling *base priority* to each thread. This *base priority* is determined by the combination of the priority *class* of its process and the priority *level* of the thread. A thread can have any of the following seven priority levels: IDLE, LOWEST, BELOW_NORMAL, NORMAL, ABOVE_NORMAL, HIGHEST, and TIME_CRITICAL. The *base priorities* range from zero (lowest priority) to 31 (highest priority).

Given this priority structure, Windows NT performs priority based preemptive scheduling. When two threads have the same *base priority*, a time sharing approach is used. REALTIME priority class threads have non-degradable priorities, while NORMAL and HIGH priorities can be decayed by the NT scheduler.

However, at a fundamental level, Windows NT was designed as a general purpose OS and many of the policies/mechanisms are geared towards optimizing the average case, and this is at odds with the high predictability requirements of many real-time environments. The following are some of the limitations in Windows NT – due mostly to the lack of provisions that take into account the priority of an event/object by various services/mechanisms – that may contribute to unpredictable delays for user applications [8, 11, 14].

The priority level of interrupts is always higher than that of a user-level thread, including threads in the real-time class. When an interrupt occurs, the trap handler saves the machine's state and calls the interrupt dispatcher. The interrupt dispatcher among other things, makes the system execute an Interrupt Service Routine (ISR). Only critical processing is performed in the ISR and the bulk of the processing is done in a *Deferred Procedure Call* (DPC).

DPCs are queued in the system DPC queue, in a First In First Out (FIFO) manner. While this separation of ISR and DPC ensures quick response to any further interrupts, it has the disadvantage that the priority structure at the interrupt level is not maintained in the DPC queues. A DPC is not preemptable by another DPC, but can be preempted by an (unimportant) interrupt. As a result of all this, the interrupt handling and the DPC mechanisms introduce unpredictable delays both for interrupt processing and for real-time computations. More generally, the lack of provision for avoiding priority inversions is the primary problem for real-time applications. Windows CE, another operating system from Microsoft designed for embedded communication and entertainment applications, supports priority inheritance (See Section 5).

Threads executing in kernel mode are not preemptable by user level threads and execute with dispatching temporarily disabled, but interrupts can occur during the execution of the kernel. As kernel level threads can mask some or all interrupts by raising the CPU's current IRQL (Interrupt request levels), the responsiveness at any point in time to an interrupt depends on the mask set by kernel entities at that time, and the execution time of the corresponding kernel entities. Also, since only kernel level threads are allowed to mask and unmask interrupts, even an unimportant interrupt can adversely affect a real-time priority user level thread. All of these do not bode well for real-time processing.

Unpredictability also occurs because some system calls (e.g., some GUI calls) are synchronous and are executed by system processes running at a non-real-time class priority.

Given these, the natural question to ask is: What are the conditions under which NT can in fact be used for real-time applications? This is what we intend to explore in the rest of the paper, first by studying the behavior of the real-time related NT components and then by using them in a prototype real-time application.

3. Real-Time Features of Windows NT

In this section, we report on the results of experiments conducted to evaluate real-time features of Windows NT Workstation 4.0.

The platform used was a PC equipped with a 233MHz Pentium processor, 64MB of RAM and 256KB of cache. Where communication is called for, the network involved was a 10Mb Ethernet. Each PC uses 3Com3C590 combo Ethernet card connected via department wide network. Timing of events and the time taken for various activities was determined using NT's `QueryPerformanceCounter()`, a counter with a a resolution of 0.83 μseconds.

3.1. I/O Interference on Real-Time Threads

Our experiments were targeted towards the behavior of threads at REALTIME priority class and their effect on the

I/O Subsystem, and visa versa. To this end, we used two threads both with same thread priority in the REALTIME class, one performing I/O and another CPU-intensive thread performing a continuous `For` loop. The following three experiments were conducted:

Experiment 1: To study the effect on keyboard and mouse I/O, the I/O thread was made to read from the keyboard/mouse. When the CPU-intensive thread was running, it was observed that no I/O activity took place. After the CPU-intensive thread completed, all the keyboard inputs were processed. This shows that the CPU-intensive real-time thread essentially shuts out keyboard/mouse I/O even when this I/O occurs from/to a real-time thread.

Experiment 2: To study the effect on disk I/O, the I/O thread was made to write a file, specifically, 40,000 64-bit values were written. The time-stamps for the I/O and CPU-intensive activities were found to be interleaved indicating time-sharing between the two threads. This shows that a CPU-intensive real-time thread did not shut out disk I/O.

Experiment 3: To study the effect on network I/O, the I/O thread was made to read data from a remote server using Windows Sockets API. Again here, the time-stamps for the two activities were found to interleave indicating time-sharing between the two threads. This shows that a CPU-intensive real-time thread has no adverse impact on network I/O.

To explain the above observations, we must briefly explain how NT handles I/O – beyond the use of DPCs. In the Windows NT I/O subsystem, I/O requests pass through several stages of processing:

1. The I/O manager sends the request in the form of an I/O request packet (IRP) to the device driver. The driver starts the I/O operation.
2. The device completes the I/O operation and interrupts. The device driver then services the interrupt. (This involves execution of ISR and queuing of a DPC.)
3. The I/O manager completes the I/O request.

In the third step of I/O processing, the system writes the data from the I/O operation into the address space of the thread which requested the I/O. In this step, two mechanisms are used [1]:

- *Buffered I/O*: Used for slower I/O devices where the data transfer first takes place into the system memory area and an Asynchronous Procedure Call (APC) is queued to copy this data into the user thread' s local area.
- *Direct I/O*: Used for faster devices like the disk. The data transfer directly takes place into the user thread's local address space, which is locked by the system.

In Experiment 1, since the keyboard and mouse I/O are performed as *Buffered I/O*, the execution of APCs responsible for copying of the data into the user thread's address space was not possible until the CPU-intensive thread was completed. This is because the input processing for keyboard/mouse is actually done by threads in the kernel, which are not running with real-time priority. Since priority inheritance is not being done, the threads processing input are not executed until the CPU-intensive thread completes.

On the other hand, for Experiments 2 and 3, since the disk and network I/O are performed as Direct I/O, the system locks the corresponding threads' buffer space into memory. This ensures that the I/O is performed even if a CPU-intensive real-time thread of the same priority is running, which is possible due to time sharing between threads of the same priority.

Win32API Function Name	**Time (μsecs)**
`CreateProcess()`	2600
`SetPriorityClass()` from normal to real-time priority class	240
`SetPriorityClass()` for all others combinations	125
`SetThreadPriority()` for a thread to set its own priority	9
`SetThreadPriority()` for a thread to set priority of another thread of the same process	10
`QueryPerformanceCounter()` to obtain the current time-stamp	6

Table 1. Time Taken for System Operations

3.2. Time Taken for Process/Thread System Calls

In order to perform user-level scheduling of real-time threads, which we consider as a possible approach to run real-time applications on Windows NT, we conducted experiments to find the time taken for the completion of various process/thread related Win32 API calls. The values obtained from our experiments are listed in Table 1. The times listed (in μsecs) are times that fall within the 90th percentile, i.e., 90% of the 1000 observations had values that were equal to or less than the reported number. We are reporting this and not worst-case (or averages) because of our interest in soft real-time and not hard real-time (or timesharing) applications.

3.3. Time Taken by System Activities

The next set of experiments was done to identify the system activities taking place in the background and worst case processor time needed to perform these activities. To this end, we observed the system without any other application running and logged the activity every second for a continuous period of 30 minutes. The following were (individually) observed in at least one of the logs.

- Process 'system' had 23 threads, thread1 getting a maximum of 53ms.
- Process 'csrss' had 10 threads, thread4 getting a maximum of 50ms.
- Process 'services' had 18 threads, thread15 getting a maximum of 50ms.
- Process 'perfmon' had 2 threads, thread1 getting a maximum of 53ms.

Other threads of these processes consumed negligible processor time. Even though the above were not observed within the same second, the observations mean that system activities can take at most a total of 153ms in a one second interval (discounting the time Process 'perfmon' takes since this is a performance monitor we instantiated, not a system activity).

It should be noted that when user processes are running in the system they may generate some system activity such as page-faults.

3.4. Summary and Recommendations

From a real-time perspective, the above observations imply that to use Windows NT for real-time applications, the following principles should be practiced:

(a) Lock pages in memory for real-time threads. Ensure that real-time threads are not inactive for a long time, since NT may unlock pages of inactive threads.

(b) The potential blocking time due to NT system activity must be taken into account when accounting for the delays incurred by an application thread.

(c) If there are processes or threads doing network or disk I/O, the effect of systemwide FIFO DPC queues may lead to unbounded response times even for real-time threads. If the duration of I/O activity in a given period can be characterized, it may be possible to pessimistically compute the response times.

(d) One should not depend on the Windows NT scheduler to accomplish the correct "fair sharing" behavior in cases where screen, keyboard and mouse interactions are at the same level of priority as the other real-time CPU-intensive tasks.

(e) To achieve more predictability for real-time tasks in general, and to achieve responsiveness for operator/human inputs in particular, a real-time system designer must not design the system such that real-time threads monopolize the CPU and I/O all the time. One must leave some computation and I/O time for those important but non-real-time NT tasks, such as those servicing the interactive I/O activities, to execute. These non-real-time NT tasks are not under our (user-level) control, but will have adverse effect on the intended real-time tasks if not executed in time. To accomplish this goal, one approach is to use periodic execution with user-level controlled cooperative preemptions, i.e., to design all threads in the real-time class as periodic tasks using a heartbeat timer mechanism described in the next section, such that real-time threads voluntarily give up the CPU to allow interactive I/O operations to complete.

4. Evaluating NT in a Real-Time Setting

To understand NT better, we prototyped a real-time control scenario involving multimedia information. In particular the focus was on the *operator's workstation* [9]. The software running on the workstation has the following components [10].

Operator input: The operator inputs control messages and actuator settings. An input has to be recognized, processed and sent to a remote destination through the network. The control message is processed at the remote node and the necessary control action is taken. After this an acknowledgment will be sent back to the operator station.

Incoming sensor data: Data arrives from sensors at regular intervals and must be stored in a ring buffer in main memory. A consumer process reads a single record from the buffer, performs some computation, and displays the result on the screen in a graphical format.

Incoming video streams: Also executing at the operator workstation is one video process responsible for retrieving streaming video from the network and displaying it on the screen. It is reasonable to assume that such a software will most likely be COTS (commercial off the shelf).

4.1. Design

The following general principles were followed in designing the prototype application:

- *Efficiency through threads*: For reasons of efficiency we attempted to use threads wherever possible, and processes elsewhere.
- *Achieving periodicity functionality*: Some of the processing, e.g., sensor data processing, is periodic. We achieved this periodicity by implementing a heartbeat timer. This is a process running at the highest real-time priority. It periodically sets events on which different processes or threads wait. Each time an event is set, the corresponding thread or process can execute one periodic instance and then once again wait for the event to be set. Currently, the heartbeat timer uses the `Sleep()` call to suspend itself till it is time to signal the next event.

In addition to designing the operator workstation, there was a need to model the entity with which the operator interacts. This entity may correspond for example, to the local controller on the factory work floor, which actually carries out the operator's instructions, monitors the local state and sends state information back to the operator. Such an entity was modeled as a *remote server*. UDP was used for communicating between operator station and the remote server. The source of the video stream was just another node on the network to which the operator workstation is connected.

Besides the *Heartbeat timer*, the main entities at the remote server are:

Remote producer: This periodically generates sensor data and sends it to the operator workstation.

Remote operator input process: This entity waits for operator input data from the operator workstation. Currently, this acknowledges the data by sending the same data back to another thread on the operator workstation.

Besides the *Heartbeat timer*, the main entities at the operator workstation are:

Receiver: This is a periodic process that receives sensor data from the remote server. It then stores the same data in a circular buffer.

Consumer: This is a periodic process with the same period as the receiver. It reads sensor data from the circular buffer and stores it in memory. A precedence relation exists between the receiver and consumer, enforced using an event which the receiver has to set to allow the consumer to proceed. The buffer management protocol has the following retrieval semantics: The consumer always receives the most recent data. Many consumers will require this type of semantics. For example, the operator is interested in the most recent speed measured within a turbine. However, the implemented buffer data structure is general enough to accommodate a wide variety of retrieval semantics.

Operator input process: This process waits for the operator to provide commands and sends them out to the remote server.

In order not to be affected by NT's inability to handle mouse/keyboard processing and screen displays in a timely fashion, all of the operator interactions with the system were simulated via memory reads/writes. Specifically, operator input was implemented by reading 1K bytes of information from a specified memory location. At the end of this section, we discuss approaches to accommodate such operator interactions and experiment with one possible approach.

Operator ack process: This waits for acknowledgments sent by the remote server in response to operator input messages. It stores the received acknowledgments in memory.

4.2. Implementation Details

At the operator workstation, a single process *clmain* is launched. It initializes global events on the operator side and spawns 2 processes:

1. Heartbeat timer.
2. Main operator process. This, in turn, spawns 4 threads corresponding to receiver, consumer, operator input and operator ack processing entities.

At the remote server a single process *remmain* is launched. It initializes global events on the remote server and spawns 2 processes:

1. Heartbeat timer.
2. Main remote process. This, in turn, spawns 2 threads corresponding to remote producer and operator input processing entities.

The heartbeat timer on each side is also used to control various housekeeping activities and to terminate the different threads and processes gracefully.

Probes are inserted into the code for taking timing measurements. As before, the high frequency `QueryPerformancecounter()` function is used by the probes for these measurements. In-memory logs are maintained of the various events and their timing. At the end of the experiment, the logged data is written to disk by the parent process on each side.

4.3. Experimental Setup

The following processes were running for this set of experiments.

- Operator workstation entities.
- Remote server entities.
- Realvideo player, transmitting distance learning course material. [1]

We observed that when launched, the video player spawns a few threads which run at various priorities in the HIGH priority class. This type of priority enhancement is used by COTS applications to boost their performance. Using the performance monitor to observe the player's load indicated that there are occasional bursts of activity during which the player uses up to $70 - 90\%$ of the CPU. This is perhaps due to the local buffers being filled after a network congestion. Each experiment was run with the video stream started at roughly the same place. This was to ensure that the workload was comparable.

In all the experiments, the entities at the remote server side were run in the REALTIME priority class:

- The priority of the heartbeat timer is always set to HIGHEST level within the REALTIME priority class.
- The priority of remote producer and operator input processing entities are set to the NORMAL level within the REALTIME priority class.

This was done to allow the measured delays to reflect activity at the operator end and not be affected by delays due to other activities at the remote server.

At the operator workstation the priority of the processes was varied between the NORMAL, HIGH and REALTIME classes and the priority of all threads was set to NORMAL.

[1] Note that we have no control over the Realvideo player, including what threads it spawns and what priorities these threads execute at.

4.4. Experimental Results

Of interest to us are the timing properties of the three types of processes – video display, sensor data processing, and operator command processing – as a function of the priority level at the operator side, the offered load, and the size of the data shipped by the operator. As for the workload tested, what we have experimented with is a real work load [10] characterizing a typical operator workstation which experiences periodic sensor data input in 1K byte sizes, sends out sporadic operator commands not more often than every 100ms, and displays one video window.

The quality of the video output was used as an indicator of the effect on the video player's performance. However, we saw no perceivable differences in the quality of the video output as we experimented with different parameters. Similarly, in all cases, the sensor data processing (by the receiver-consumer pair) was not affected. This pair was found to execute at the specified frequency with almost no jitter. So we focus on the *Round Trip Time* (RTT) as seen by the operator input process.

4.4.1 Round Trip Delays

In this set of experiments, operator input and receiver-consumer entities run at the same frequency. Figures 1, 2, 3 respectively plot the round trip delays experienced by 1000 1KB operator commands for operator input frequencies of 30, 50, and 100ms. Also, some important measures (maximum, mean, variance) of the round trip delays are shown in Table 2. These clearly indicate that:

- The round trip delay for the operator input vary much more if the operator input processes are at NORMAL priority. There is significantly less variance if HIGH or REAL_TIME priorities are used.
- Even through the average round trip delay is very similar in all cases, the average decreases as priority increases. For example, for a operator input period of 50ms, the average round trip delay ranges from 3.356ms corresponding to NORMAL priority to 2.428ms for REALTIME priority.
- The maximum value also tends to decrease with increasing priority.
- The most dramatic change is in the variance. For instance, for a 50ms operator input period, it decreases from 19.721 (normal) to 5.879 (high) to 0.242 (real-time). This augurs well for predictability.

The above results indicate that even with all the processes running, the system has enough processing power to handle all the tasks. The large variance at NORMAL priority, therefore, can be at least partially attributed to the Realvideo player as well as system activities which are running at a higher priority. As the process priority is increased, this effect diminishes, resulting in lower variance. The real-time processes in the prototype are typically not computationally intensive, but have strict delay requirements. System activities on the other hand can suffer some amount of jitter. As such, it makes sense to elevate the priority of these real-time processes to ensure that their delay requirements can be met.

Priority	Period (ms)	Max. (ms)	Mean (ms)	Variance
Normal	30	75.297	2.654	8.846
High	30	27.938	2.523	2.571
Real-Time	30	16.583	2.362	0.260
Normal	50	43.759	3.356	19.721
High	50	44.098	2.673	5.879
Real-Time	50	9.475	2.428	0.242
Normal	100	54.858	3.104	16.523
High	100	32.740	2.765	5.184
Real-Time	100	2.884	2.379	0.004

Table 2. Statistics for round trip delay.

4.4.2 Better Response for Operator Thread

To exercise the system even further, we experimented with different message lengths (corresponding to differing amounts of operator input) and a higher rate of operator input (needless to say that an operator input every 10ms is humanly impossible, but helps stress the system).

In the experiments discussed thus far, the threads in the operator and remote processes had the same priority level (NORMAL). So, these threads ran in a timeshared fashion. With a view to improve the responsiveness to operator threads we experimented with an alternative approach whereby as soon as the operator input thread was given a ticket to run, the priority of the operator input and acknowledgment was raised to HIGHEST, thereby giving them a priority higher than the sensor data threads. This ensures that the operator input and acknowledgment are processed with a higher priority than the sensor data threads. Table 3 gives the RTTs for these experiments. (Column "Size" refers to the size of the message containing operator input/commands.) It was observed that there was no effect on the timeliness of sensor data threads.

Size (bytes)	RT Prio.	Max (ms)	Mean (ms)	Variance
1K	Normal	9.473	2.496	0.69
1K	Highest	6.990	2.445	0.143
1.5K	Normal	10.735	3.495	0.2
1.5K	Highest	7.211	3.351	0.025
2K	Normal	14.15	4.738	0.693
2K	Highest	6.043	4.212	0.014

Table 3. Statistics for RTT with 10ms period.

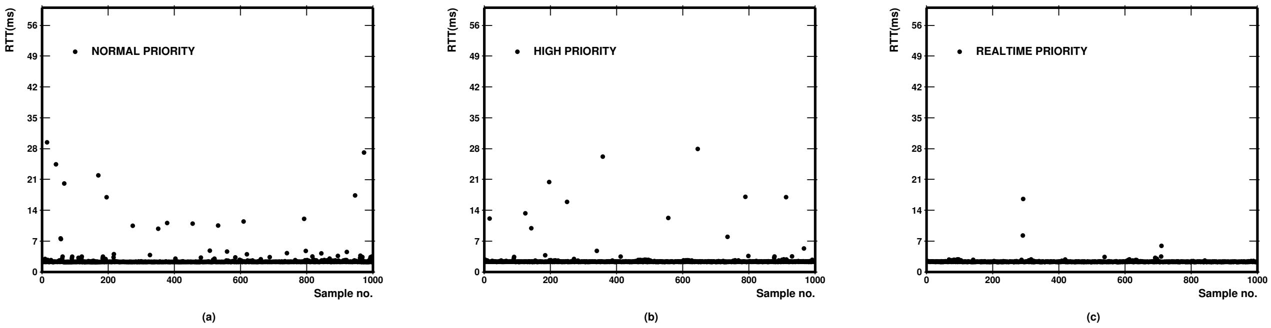

Figure 1. RTT with operator commands every 30ms.

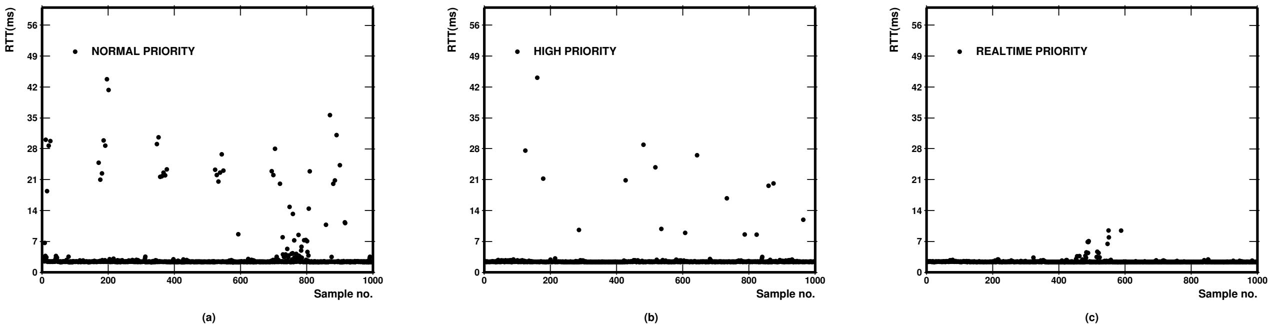

Figure 2. RTT with operator commands every 50ms.

The maximum delay was considerably lower when we increased the priority of the operator thread to the HIGHEST level from NORMAL. In fact, because the maximum is larger than the period when NORMAL priority level was used, some of the messages miss their deadlines. Specifically, we found that for a message size of 1.5K, and NORMAL priority, there were 2 tardy messages and for a message size of 2K there were 4 tardy messages out of 300 messages. No tardy messages were observed when the priority was raised to HIGHEST level.

Even though the mean delays do not differ by much, the variance becomes very small when the operator thread's priority is increased to HIGHEST. For example, for a 1.5K message size, an order of magnitude improvement is seen.

These experiments suggest that by systematic manipulation of thread priorities, better and more predictable response times can be achieved.

4.4.3 Accommodating Interactions with Operator

Since no deadlines were missed even with the period set to 10 msecs for both operator and sensor, we decided to evaluate the effect of allowing keyboard/screen I/O from the operator thread. To this end, we made the operator thread display the acknowledgment from the remote server on the operator's screen. This experiment was done with a message size of 1K and as described earlier, the whole message is returned as the acknowledgment, and is then displayed to the operator. Results (see Table 4) indicate that when the period is set to 10 or 20ms, because screen I/O is handled at lower priorities, many of the deadlines are missed and this has a cascading effect. However, increasing the period of the operator thread to 100ms alleviates these problems. At this period, even with screen I/O, no deadlines are missed and the variance in the round trip delay is low. These results are very encouraging in that, given human response times, operator interactions are likely to occur at relatively low frequencies, i.e., higher periods. So it should be possible to accommodate them in many situations. Characterizing such situations is one of our next steps.

Period (ms)	Max. (ms)	Mean (ms)	Variance
10	126.873	6.150	93.12
20	52.278	3.55	14.067
100	30.913	2.615	1.81

Table 4. Statistics for RTT with screen I/O.

An alternative to assigning or requiring higher periods for operator threads is to permit controlled preemption of different real-time threads so that operator interactions can take place. We plan to explore this avenue also.

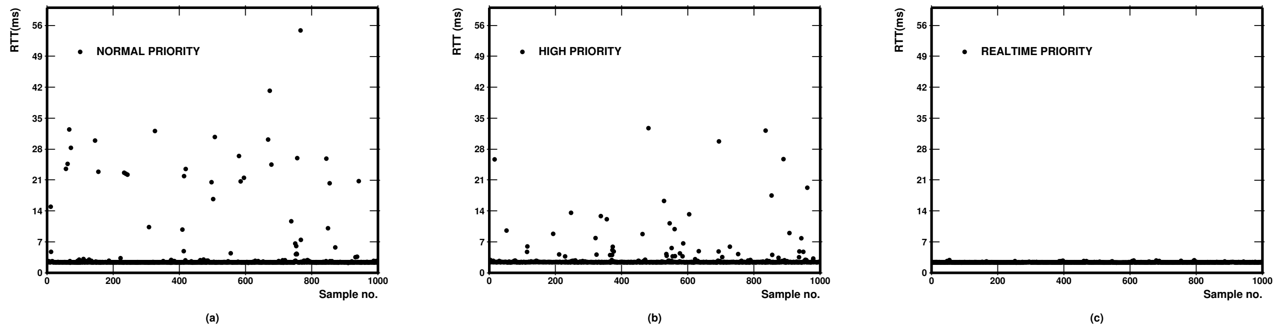

Figure 3. RTT with operator commands every 100ms

4.5. Summary of Results

The prototype implementation models a simple multimedia operator workstation for factory operations. The prototype is parameterized, so different configurations and workloads can be tested. In particular, the frequency of the operator input as well as the receiver-consumer pair are tunable parameters, as is the message size. A simple heartbeat timer mechanism along with events was used to emulate periodic processes. This heartbeat timer is an ideal candidate for implementing meta-level scheduling functionality in Windows NT. Even without using specialized scheduling, it was observed that just using HIGH or REAL_TIME priority significantly reduces the variability in response times, without any observable degradation in system performance. This suggests that as long as the application tasks do not monopolize the CPU for long durations, and there is sufficient CPU capacity, using these priority assignments may be sufficient to meet the performance requirements of these processes – even when I/O is involved. However, if the periodic workload per process is high, or if a process is a COTS application whose workload varies, it will be necessary to impose some additional controls on the amount of time allocated to a task. We are currently exercising the different elements of the prototype to experiment with these situations in order to evaluate several scheduling approaches to meet timeliness requirements. We are also evaluating different approaches to allow keyboard/screen/mouse I/O under controlled conditions.

5. Other Approaches towards Predictable NT

Our goal in this paper has been to understand NT's capabilities and the extent to which NT can be used judiciously as is by being "careful" while exploiting NT's capabilities.

We now discuss other approaches that can be envisaged for overcoming the limitations of NT [8].

- Use a highly constrained and fine tuned Windows NT and application environment. This approach is suitable mainly for applications whose worst case resource behavior can be determined beforehand. The solution involves using a restricted set of support from Windows NT, and using careful analysis to reduce the timing unpredictability. Once such an application is implemented, changes are either disallowed or require a complete overhaul in order to guarantee the timing requirements.
- Modify the Windows NT kernel [4]. This option requires continuous changes to the modifications as new versions of NT appear.
- Couple real-time operating systems with Windows NT, with each OS running in a different machine.
- Provide a Win32 API wrapper around a real-time OS. This alternative does not meet the requirement of running Windows NT applications in unison with the real-time tasks.
- Modify the Hardware Abstraction Layer (HAL). This is only a partial solution, which needs to be coupled with a small real-time executive as is done in Venturcom's RTX [14] and Real-Time Linux [13].
- Compose a Windows NT driver to run all the time critical threads. This is the approach taken in [6]. This approach has the drawback of (1) having to create a totally new API for users to construct real-time tasks to run in this driver environment, and (2) potentially still incurring all potential blocking from other ISR and DPCs.

The above do not appear to be adequate or feasible for our purpose. In contrast, many commercial extensions to NT for real-time complement Windows NT kernel with a real-time kernel. There are two alternatives. In the first alternative, the real-time OS co-exists with Windows NT. In the second alternative, the real-time kernel becomes part of Windows NT in the form of a device driver. The objective of the device driver is to provide the services which are not supported by NT and to trap hardware interrupts.

The following is a description of commercial products using these approaches:

RadiSys [8] places INtime, a real-time kernel based on iRMX, outside the Windows NT address space. The two

OSs are able to communicate via an extension to the Windows NT API. This mechanism uses hardware support in order to achieve a complete separation between two co-resident operating systems. This approach is similar to the work done by Jeffay and Bollela [3]. The latter work implemented a small CPU executive which multiplexes the CPU between their own real-time kernel and an IBM microkernel with an OSF1 server. The difference between INtime and [3] is that INtime runs Windows NT as its lowest priority process and in [3] the CPU is shared between the two operating systems. The measures adopted to impart predictability include:

- The INtime real-time kernel is based on the iRMX kernel, and unlike NT, application threads and interrupt handlers share the same priority structure, allowing suitable priority assignment to threads and interrupts.
- INtime runs Windows NT as its lowest priority process. Real-time interrupts and active real-time threads immediately preempt any running NT thread, and also disable all non-real-time interrupts. Assuming that iRMX implements some sort of priority inversion handling protocol, this arrangement should be able to alleviate the unpredictable delays (at least for the real-time threads).
- The HAL has been modified to ensure that interrupts reserved for real-time use are never masked. The HAL trappings attempt to assign interrupt handlers to interrupts reserved for real-time kernel use.
- RadiSys also claims to maintain complete address space isolation and memory protection between real-time and Windows NT processes.

Anther commercial solution is the one offered by *Imagination Systems* [2]. Their real-time subsystem called *HyperKernel* has its own scheduler, its own set of services, and its own internal kernel. From the literature, it seems that they are using hardware support to achieve a complete separation between two co-resident operating systems and also do not modify the HAL. Also, it seems that interrupts are controlled by direct access to the Interrupt controller. Superficially at least, their methodology appears to have elements of commonality to the approach taken by RadiSys as well as the work by Jeffay and Bollela [3].

The *VenturCom* approach [14] places a real-time OS (RTX) as a subsystem inside Windows NT in the form of a device driver. RTX facilities include priority scheduling, non-degrading priorities, inversion management, IPC support, fast clocks and timers, memory allocation and page fault elimination. We believe that NT is also considered the lowest priority process, but have not been able to confirm this. The approach taken by VenturCom is similar to the one taken in Real-Time Linux [13]. Here a software emulation of the interrupt control hardware interacts with the OS. Interrupts directed to Linux are passed to the emulation software after real-time tasks are executed by a small real-time executive. The VenturCom approach also modifies the HAL in order to isolate interrupts between NT and the real-time subsystem.

Although these techniques may considerably ameliorate the unpredictability of standard Windows NT, potential problems arise due to changes they entail to NT or HAL as well as due to the additional API or Operating System that one needs to master. Most of these products are still in the development or initial deployment stage and more details of their approaches are needed for in-depth evaluation and understanding.

Another alternative to NT is Windows CE [12] that is targeted at embedded applications.

Unlike Windows NT, Windows CE does not provide process priority classes. At any time, there can be a maximum of 32 processes within the system. Within a process, threads can have any one of the 8 possible thread priority levels: TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, ABOVE_IDLE and IDLE. The level ABOVE_IDLE is not present on Windows NT. Time-critical threads have the highest priority and are not time shared. Whereas Windows NT dynamically adjusts its priorities for threads in dynamic priority class, Windows CE does not change the priorities of *any* of the threads dynamically – this may be detrimental to non real-time computations that compete for processing resources. Windows CE does modify the priorities to handle priority inversion. As noted earlier, Windows NT does not support any priority inheritance, not even at real-time priority class.

Paging activity in Windows CE uses the above mechanism to avoid priority inversion. Normally, the kernel thread handling page faults runs at priority level NORMAL. When a thread with priority level higher than NORMAL suffers a page fault, the priority of the kernel paging thread is raised to the priority of the thread causing the fault. This ensures that a thread is not blocked by another lower priority thread even if it suffers a page-fault. Also, a thread running at the highest priority can be guaranteed to execute within a known period of time. This is not possible in Windows NT as some of the system threads are not run at inherited priority (as observed in our keyboard I/O experiment at real-time priorities). Unbounded delays due to system wide DPC queues (observed in Windows NT) are avoided as priority inheritance is applicable to each and every system thread including the paging activity thread. Windows CE queues waiting threads in priority order rather than simple FIFO as in Windows NT. Each thread priority has a different FIFO queue.

Interrupt processing in CE is done in two steps:

1. ISR : performs minimal processing, runs at higher priority than ISTs.

2. IST : Interrupt Service Thread, these usually run at top two priority levels, ensuring that they run quickly.

Nested interrupts are not supported. (i.e. no interrupt is recognized when ISR runs) The time for their execution can be bounded only if it is guaranteed that the ISTs run at the same priority as the thread causing it (i.e. priority inheritance applies to ISTs). From the available literature, it seems that ISTs run at the two top priority levels.

In summary, Windows CE provides more levels of thread priorities, priority inheritance is supported, and so interrupt latencies can be bounded slightly more easily than Windows NT.

6. Conclusion and Future Work

By measuring the delays involved in NT's real-time functions we have gained a number of insights regarding the feasibility of Windows NT for real-time applications. And, by building a prototype application which models a simple multimedia operator workstation, we have demonstrated how to use these insights in the design of such real-time applications. With the use of real-time priorities, we have shown that it is possible to improve the stability of certain real-time tasks.

The one place that Windows NT is very platform dependent is how I/O interrupts are handled. A designer needs to be aware of the effect of the systemwide FIFO DPC queue on *any* user thread. This queue has priority over all user-level threads. This may lead to unbounded delays if a badly designed driver is used. Thus this is the part of Windows NT about which one cannot make general analytical or quantitative statements for all potential platforms. A user needs to know what devices and what drivers he/she is using, and what performance characteristics a particular driver will induce on the system. If it is possible to characterize various I/O activities and their contributions to the DPC queue, one can possibly place some pessimistic bound on the response time for real-time threads. Soft real-time applications which can tolerate occasional delays due to factors like systemwide DPC queues can be realized using NT. Windows CE has several improvements to correct this problem, through priority inheritance.

To achieve more predictability for real-time tasks in general in Windows NT, and to achieve responsiveness for operator/human inputs in particular, in this paper, we offer the following key recommendation to a real-time system designer using NT:

Design the system such that real-time threads do not monopolize the CPU and I/O all the time. Some computation and I/O time must be left for executing important but non-real-time NT activities, such as those servicing interactive I/O. These non-real-time NT tasks are not under our (user-level) control, but will have adverse effects on the intended real-time tasks if not executed in time. To this end we experimented with one approach in this paper: facilitating periodic execution with user-level controlled cooperative preemption. So, all threads in the real-time class were designed to execute periodically using a heartbeat timer mechanism such that real-time threads voluntarily gave up the CPU to allow interactive I/O operations to complete.

We plan to implement a systematic rate-based user-level scheduler to schedule tasks in a given real-time environment. The prototype system described in this paper provides a suitable test-bed tool to experiment with different user-level scheduling schemes. The heartbeat timer can be effectively used for such user-level scheduling.

7. Acknowledgments

Our thanks to John Howard for his careful review of a previous version of this paper and for his valuable suggestions.

References

[1] H. Custer, "Inside Windows NT", *Microsoft Press*, 1993.

[2] HyperKernel, *http://www.imagination.com/*.

[3] G. Bollella and K. Jeffay. "Support for Real-Time Computing within General Purpose Operating Systems", *In Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Chicago, IL, May 1995, pp.4-14.

[4] Z. Deng and J.W.S. Liu. Scheduling Real-Time Applications in an Open Environment. *18th Real-Time Systems Symposium*, Dec 1997, pp. 308-319.

[5] O. González, C. Shen, I. Mizunuma, and M. Takegaki. Implementation and Performance of MidART. In *IEEE Workshop on Middleware for Distributed Real-time systems and Services*, San Francisco, CA, December 1997.

[6] I. Kawakami, Y. Katayama, and H. Kurosawa. A Newly Structured Real-Time Control Mechanism based on Personal Computers(RT-PC). In *International Conference on Electrical Engineering*, Matsue, Japan, July 1997.

[7] I. Mizunuma, C. Shen, and M. Takegaki. Middleware for Distributed Industrial Real-Time Systems on ATM Networks. In *17th IEEE Real-Time Systems Symposium*, December 1996.

[8] "Approaches to Real-Time Windows NT", *http://www.radisys.com/products/rtos/intime_app.html*.

[9] C. Shen and I. Mizunuma. RT-CRM: Real-Time Channel-Based Reflective Memory. In *IEEE Real-Time Technology and Applications Symposium*, June 1997.

[10] H. Shimakawa, Mitsubishi Electric Corporation, personal communication, 1996.

[11] M. Timmerman, "Windows NT as Real-Time OS", *Real Time Magazine*, 1Q97 Issue, *http://www.realtime-info.be/encyc/magazine/articles/winnt/winnt.htm#backfive*.

[12] Windows CE web pages: http://www.microsoft.com/windowsce/

[13] V. Yodaiken, "A Real-Time Linux", *http://luz.cs.nmt.edu/ rtlinux/*.

[14] VenturCom home page: *http://www.vci.com/*.