An evaluation of Windows NT with respect to real-time capabilities in the context of the QUITE project

Philippe Bernadat
The Open Group
bernadat@opengroup.org

May 12, 1999

Summary

It is widely agreed that NT would not be a reasonable choice as an RTOS for hard real-time control. For critical hard real-time environments a small and predictable RTOS coupled with carefully designed dedicated H/W drivers is chosen in the vast majority of cases, even though it is rarely formally proven that the level of predictability is sufficient. The functionality and services provided by such RTOS are minimal, more general purpose computing, data transformation, user interfaces are often performed on a separate host.

Soft real-time applications may accommodate to run on less predictable systems, or what we can also refer to as GPOS (general purpose operating systems) provided that one can acquire enough confidence on the overall behavior, by means of empirical measurements. The two obvious reasons to choose a given GPOS are 1) it offers a rich set of services and standard APIs, 2) it is widely available on off-the-shelf hardware. NT is clearly such a GPOS.

In the context of the QUITE project we analyze how realistic it is to use off-the-shelf NT, how we can circumvent its deficiencies. As our analysis will point out, the real-time behavior of NT is primarily dependent on the H/W device drivers. It is therefore impossible to state that NT fulfills our needs, even for a given processor speed. Along with the analysis, one of our objective has been to provide a set of tools that one could use to evaluate its own host environment, without any requirement for dedicated hardware measurement device (timer board or such).

To conciliate both hard real-time requirements and use of GPOS services, commercial companies have developed real time extensions to NT, which basically consist of an aside real-time OS or SubSystem, sharing the processor with NT but with higher scheduling privileges. The RT extension tasks communicate with the NT ones by means of a message passing facility and shared memory. We explain to what extent such extensions could improve predictability.

Table of Contents

1. Preamble

Each one of the 60 or so measurements performed for this study last from 15 minutes to 12 hours. It was our intent to systematically run each single test for at least a day on two distinct platforms but it quickly became unrealistic given our hardware resources. Nevertheless, the tests are designed to run for such long times without any overflows and it is only a matter of time and hardware resources to perform longer measurements. We are in the process of running all tests for 12 hour periods on our reference platform in order to obtain fully comparable figures and to better quantify the outliers.

2. Context and background

The overall objective of the QUITE project is to build a large-scale, QoS-aware, real-time distributed system. The system middle-ware is based on DARPA funded research technologies, namely TMO, AQuA, EPIQ, CEDAR, MSHN, DeSiDeRaTa, QUASAR: SWIFT, TAO, HPF, ViewNT, Ensemble, NetSimQ, ASSERT, Darwin.

Without entering into the details of each of the components, from the OS's perspective, the fundamental property of such a system is the end-to-end QoS dimension, requiring that the system services be predictable from the network devices up to the middle-ware node inter-connection management tasks. This implies predictability through the networking protocol stack as well as more traditional requirements such as preemptible scheduling or accurate timers.

The statement of work for the QUITE contract states: *"The contractor shall integrate operating system extension components into reference implementations. The base commercial operating system for Quorum shall be Microsoft Windows NT but other commercial operating systems may also be involved. The contractor shall modify the extension components as required to assure proper interface with and operation within the reference implementation infrastructure and provide these modifications back to the developer of the extension component."* In other words there is a strong incentive to use standard off-the-shelf NT because it is widely available and its development environment and APIs are well understood.

NT's dominant role in the field of Operating Systems has naturally prompted numerous experiments and studies regarding its real-time capabilities [3], [14], [4], [17], [18]. Microsoft claims that NT is suitable for most real-time applications, as long as they can be characterized as "soft" real-time ones[1]. A real-time application is often characterized as hard when a missed dead-line has disastrous effects, ranging from non recoverable financial costs to loss of lives.

These performance studies, have demonstrated that NT's predictability is low, that the interrupt handling has only been designed to be efficient (fast) in the vast majority of cases. In this study we do not try to demonstrate that NT is viable for real time, but rather explain where its architecture conflicts with real time constraints, provide a set of tools to measure the impacts, along with the actual figures for a given platform.

The requirements frequently agreed on for a commercial OS to enable real-time determinism are:

- Preemptible and multi threaded. This is the case for NT, but interrupt processing is usually not performed in the context of a thread.
- Multiple priority levels, with a fixed priority scheduling policy, and preferably FIFO (versus Round Robin). NT does not offer a FIFO policy, the scheduling quantum is an unescapable parameter.
- Deterministic thread synchronization and switching. This is not the case for windows NT, mostly because of the interrupt processing architecture. Also the NT kernel does not handle priority inversions (neither inheritance nor ceiling), it would have to be implemented as a library with a non negligible overhead.
- Deterministic interrupt and timer services latencies. Interrupt processing in NT has been optimized for speed efficiency versus predictability.
- Wired (locked) memory. NT offers an API to lock memory, we have not checked its effectiveness.

It clearly appears that the most important obstacle to real-time control in NT is interrupt handling and this will be the primary axis of our study.

The document is organized as follows:

- A first section to explain where the scheduling and interrupt handling is an issue.
- A description of what we measure and which methodology to use.
- The measurement figures and analysis for a given platform.
- A section on the usability of real time extensions for NT
- A conclusion

3. NT Internals

In this paragraph we outline where the design of the NT kernel is an issue for hard real-time. The reader may refer to [13], [10], [11] and [1] for in depth descriptions.

As Figure 1 illustrates, the Windows NT kernel defines 4 types of runable entities. Let us describe each of them, ordered with decreasing scheduling privileges:

- **The dispatcher.** Unique entity that is activated whenever another scheduling entity blocks or when interrupts are raised. It is neither preemptible neither interruptible, but runs for fairly short periods of time.
- **Interrupt service routines (ISR)**. Activated to handle interrupts. At interrupt time, if the processor IRQL is lower than the associated driver one, the running ISR, DPC or thread will be interrupted. Control will be passed to the corresponding ISR. Typically the ISR routine will just acknowledge the hardware interrupt and queue a DPC event. It is recommended that device drivers perform minimum work at this point, but it is not enforced. ISR's are interruptible by higher level ISRs but not preemptible by DPCs or threads.
- **Deferred Procedure Calls (DPC)**. Queued by ISRs but also eventually by other kernel modules they are mostly used to perform the bulk of interrupt processing. DPCs are always processed before giving back control to threads. DPCs are interruptible (depending on the IRQL) but not preemptible by threads.
- **Threads**. There are 2 types of threads: **user threads** and **kernel threads**. A user thread belongs to a user process (or task) and shares a user virtual memory space with the other user threads belonging to the same process. User mode applications are exclusively implemented as user threads. A kernel mode thread can only be created by a kernel module (such as a device driver) and shares a unique kernel virtual space. A user thread usually runs in user mode, whereas a kernel thread always run in kernel mode. A kernel mode thread can access system privileged services such as increasing the interrupt request level (IRQL). A user thread will also run in kernel mode when it requests a kernel service, and will exclusively run kernel code at that time. Throughout the document we will refer to **user mode threads** and **kernel mode threads**. A kernel mode thread stands for either a kernel thread or a user thread executing a kernel service. User mode threads are always interruptible. Threads are preemptible according to their scheduling properties, as described in following sections.

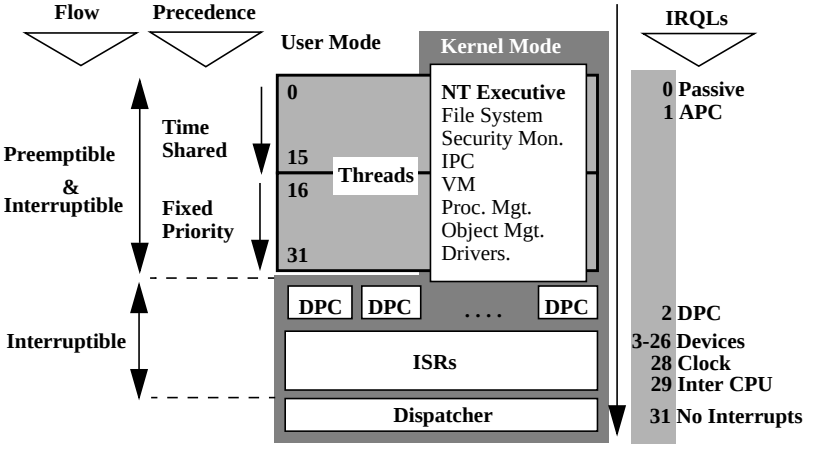

Figure 1: Windows NT Internals

The NT kernel has clearly been designed to optimize interrupt processing. A device interrupt always preempts a user mode thread, or a kernel mode thread if not masked against this particular interrupt, whichever real-time

priority level the thread is running at. This is also the case for most other OSs and it is required to determine at least the relative priority (importance) of the interrupt event but also to acknowledge the hardware interrupt. In the case of NT the bulk of interrupt processing is also considered as more critical than any other activity and the interrupt processing will complete before the interrupted thread resumes, whether or not the device driver uses deferred procedure calls. Only kernel mode threads, including the device driver threads, can raise the interrupt request level. In reality, this means that interrupt processing always has precedence over any other thread. **So it is ambiguous to state that NT is fully preemptible,** in the sense that DPCs which are in fact some sort of specialized threads are not preemptible. Interrupt processing and kernel deferred activity will always have precedence over any other activity. It is questionable why Microsoft chose not to implement DPCs as threads. There would be indeed a performance penalty, but it could be a configurable mode.

As a comparison with another micro kernel, in OSF1-MK, interrupts are queued at any time as well, but the interrupt processing itself is performed by a dedicated thread, whose priority can be adjusted according to its relative importance as compared with other real-time duties performed by other threads.

The NT kernel operates at one of 32 distinct interrupt levels. The highest ones are mapped to hardware faults, the intermediate ones are mapped to H/W device interrupts and the lowest ones are used for deferred procedure calls, and asynchronous procedure calls. The very lowest one, 0, also known as passive level, is the one at which the user threads usually run. The kernel dispatcher runs at the highest IRQ level (not interruptible) and is not preemptible. Its main task is to schedule the various runable entities (i.e. threads, DPCs and ISRs). Although 32 levels may appear sufficient, the IRQ level is not considered when queueing DPCs. Even though [12] explains how DPCs can be queued at head or tail, **it is fair to say that there is very little control on how to prioritize interrupt processing**.

The NT kernel distinguishes 32 levels of thread priority. The lower 16 ones are managed with a time shared policy and decaying priorities. The upper 16 ones are reserved for real-time threads. A real-time thread priority is fixed. The real-time threads are scheduled with a round robin policy, with adjustable quantum. The Win32 API offers a window of only 7 priority levels for a given task, but we believe that it is possible to write a kernel module (i.e. a driver) to access the full range. So we can consider that **only 16 levels of real-time thread priority and the lack of FIFO scheduling (versus Round Robin) are serious limitations.**

4. What to measure?

Let us re-state that we do not pretend to characterize windows NT since there is not one well identified NT system but instead numerous HW/WindowsNT environments. We have developed a set of measurement tools, that one can re-use on any platform provided that the processor belongs to the P6 pentium family (more precisely 735/90 and 815/100 models, or any more recent one).

The figure that is frequently required by application builders is "*How fast and with what predictability can an event be triggered and processed?* ". The event can be external, relayed by a sensor or some dedicated hardware, a timer expiration, or any kind of interrupt. This led us to develop 3 typical tests, and a 4th one more specific to our environment:

1. A preemption and interrupt disturbance test. The test analyzes how frequently and for which duration a periodic task is interrupted or preempted. This indicates with what predictability a given task (purely CPU in our case) can be accomplished, and what is the longest delay incurred. The measured activity is purely user mode CPU and does not rely on any Win32 or kernel services. This type of activity is the one that we expect to be the most predictable, since it does not depend on kernel or Win32 services. One could adapt the test to perform any desired type of activity, there are many combinations to be explored.
2. Clock and timer services test. Measures how accurate the base services are. We have found the SleepEx() call and the multi-media timers to be the most useful and accurate.
3. Interrupt latency tests. Measures the ISR, DPC and thread wake up latencies for Interrupts
4. UDP/IP latencies. This test is of much higher level but very relevant to our project environment and involves most critical layers of the NT architecture; Interrupt latencies, context switching and Win32 layer.

We think that knowledge of the test implementation is of equal importance to the resulting figures, and we will describe the tests with the appropriate level of details.

5. Methodology

Our goal is to develop tools to measure NT's real-time capabilities on standard NT workstations and servers without any requirement for dedicated measurement hardware such as special timer boards. As we expect that the figures will vary depending on the host's hardware and drivers, the measurements will need to be performed multiple times on a variety of hosts. It is therefore desirable that the tools be easy to install and run.

5.1. Measurement accuracy

On x86 PCs, Windows NT clock (GetTickCount()) and multi-media timer (timeGetTime()) services offer a standard 10 or 15 ms resolution (depending on both the H/W clock and the Windows NT variant, server or workstation). One can adjust the timeGetTime() resolution to a 1 ms minimum value through the multi-media timeBeginPeriod() service. This is not sufficient to measure sub milliseconds latencies.

As a remedy Windows NT offers a high resolution performance counter (QueryPerformancCounter()). As illustrated in table 1 this service is H/W dependent and its resolution remains low for some platforms. Additionally the service time (the time required to read the counter through the Win32 API) is up to 800 times larger than the resolution itself.

Micro-processor	1 cycle (ηs)	Counter resolution		Service time
		Freq. (MHz)	Period (ηs)	ηs
333 Mhz Pentium II	3	333	3	1611
120 Mhz Pentium	8.333	1.193	838	8125
100 Mhz Pentium	10	1.193	838	8750

Table 1: QueryPerformanceCounter() service resolution and service time

Assuming that the host platform is a Pentium based personal computer we can take advantage of the Pentium time stamp counter. Note that QueryPerformanceCounter() does use this on-chip counter for recent Pentium based platforms. The time stamp counter measures cycles. The clock frequency can easily be determined. The time stamp counter is accessible to user mode threads on windows NT 4.0, and may therefore be used to measure latencies across the kernel/user boundary. We have defined a very short asm in-line macro to access this register, using the rdtsc instruction (see the details in appendix A). This macro is accessible both to user mode thread and kernel mode threads or any kernel code. Table 2 gives a timing comparison with the standard QueryPerformanceCounter(). The service time for the macro variant is 17 to 75 times faster than the QueryPerformanceCounter() API.

Micro-processor	QueryPerformanceCounter		Inline Macro	
	Resolution	Service time	Resolution	Service time
333 Mhz Pentium II	3	1611	3	96
120 Mhz Pentium	838	8125	8.333	108
100 Mhz Pentium	838	8750	10	130

Table 2: QueryPerformanceCounter() versus inline macro timings (ηs units**).**

5.2. Collecting measurement figures

Real-time worst case measurements require long range measurement periods (hours if not days). Simply recording the worst case and/or the average figure is often not enough to study the overall behavior. On the other hand it is simply impossible to store millions of samples without impacting the measurements. We provide a library that enables aggregation of the samples as fractiles using a logarithmic indexing. Thus we minimize the memory while still capturing enough data to characterize the distribution. The library is designed so that any unexpected memory allocation required to store a sample is notified, since it may induce some noise or unexpected delay.

To reduce noise, the test results should not be displayed in real-time, but rather recorded and replayed later on.

5.3. Workloads

Since we anticipate the results to be device and device driver dependent we attempt to cover the full range of standard devices, but also concurrent cpu activity to analyze the scheduling policy influence. To characterize the various workloads we used the NT performance monitor tool, the figures are reported in table 3. The 9 types of workload are:

1. **Idle**. No user activity. The windows NT system is still connected to the network, the standard windows NT services are up and running, but there no user specific tasks are activated except the test itself. The 290 syscalls/sec are simply a result of the performance monitor probing kernel counters.
2. **CPU** periodic. A user mode CPU type work-load. This is a single instance of our preemption and interrupt test running with configurable priority attributes and configurable periodicity. The number of interrupts per second is a consequence of the test switching to a 1 ms clock granularity, to use the multi media timers.
3. **Make**. An nmake command to compile 300 lines of C code.
4. **Java**. A cpu and graphic intensive multi threaded applet. The number of interrupts per second is the result of the JDK1.2 plug-in switching to a 1 ms clock granularity.
5. **Disk**. Disk to Disk file copy. (the test copies a total of 370 Mbytes to make sure the file system cache is overflown).
6. **Network** activity. Copying files between distant hosts.
7. **Tty**. Transferring bytes from COM1 to COM2, with 115200 bits/sec settings. The number of interrupts per second in table 3 confirm that the test reaches the maximum H/W throughput.
8. **CD**. Reading files from a CD file system.
9. **Floppy**. File transfers from and to a floppy device.

When performing the measurements, the workloads are activated as a for ever loop.

Kernel counter	**Idle**	**Cpu**	**Make**	**Java**	**Disk**	**Net**	**Tty**	**CD**	**Floppy**
CPU%	0	50	85	95	8	2	24	4	1
Intr/sec	64	1026	93	1035	160	320	12860	100	73
Ctxt Switch/sec	25	100	320	1620	1500	135			60
Syst%		1	25	28	8	1.5	22	4	0.5
Syscalls/sec	290	390	4050	22000	480	300	3583	4200	370
DPCs/sec	1	1	50	1	218	230	25	30	8
Disk Read/sec			2		55				
Disk Write/sec			20		55				
Disk Read KB/s			2		2900				
Disk Write KB/s			1000		2800				
Net Packet Send/s						135			
Net Packet Recv/s						125			
Net Send KB/s						130			
Net Recv KB/s						125			

Table 3: Work-load characteristics

5.4. Target platform

Unless stated differently, our target platform is an HP XU Kayak, with the following characteristics:

- 333 Mhz pentium II
- 64 Mbyte memory.
- Seagate st34501w SCSI disk (aic78xx.sys)
- HP Ethernet Family Adapter (pcntn4m.sys)
- Hitachi CDR-8335 SCSI CD (piixide.sys)
- Windows NT 4.0 WorkStation.

6. Scheduling, Interrupts and preemptions.

This test program, measures how frequently and for how long a periodic CPU bounded program is interrupted or preempted. The test iterates over a short sequence of instructions. The code sequence includes 1) a fetch of the time stamp counter, 2) cumulating the counter delta less a base reference value (i.e. the excess duration for the code sequence) in statistical fractiles and maintaining its minimum and maximum value.

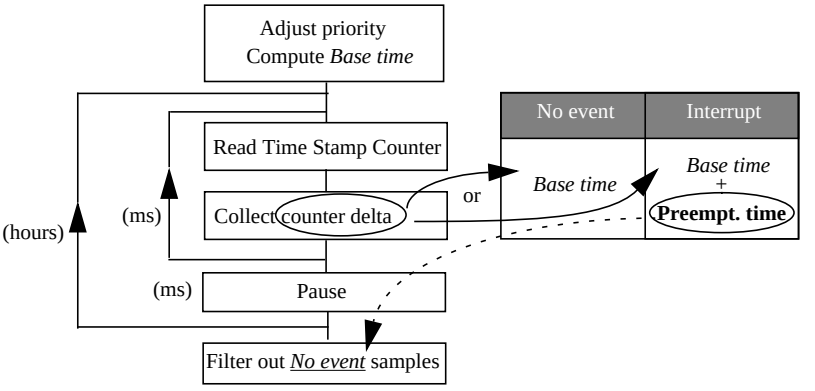

Figure 2: Interrupt and preemption measurement

The shortest time length for this sequence is measured first and considered the base reference time. The test then iterates periodically over the short code sequence, for a given number of loops. The iteration length (i.e. the active period) and the pause time (inactive period) are both configurable. The default settings are 33 ms for both. Once the test is over, the program walks through the statistics, and compares the excess duration samples with the reference value. If the ratio is larger than a configurable value (4 by default) the reference is considered as an interrupt or a preemption. After some empirical measurements, a tolerance of up to 4 times the reference appeared reasonable, the observed interrupt rate being coherent with the one reported by the NT performance monitor. Note that using a ratio of 2 or 1 does mot significantly change the results, only by a few percent as illustrated in appendix B.

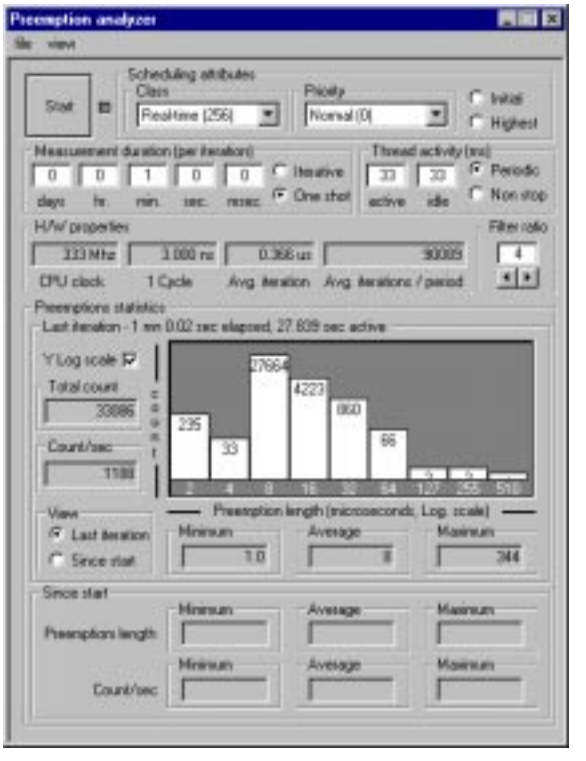

Figure 3: Preemption test graphical interface

This measurement technique is applicable only because the base reference time for the iterative sequence is relatively small as compared to the interrupt processing time. The number of cycles and corresponding time duration are given in table 4. The ISR latency stands for the minimal interrupt time for a device interrupt where the Interrupt Service Routine returns immediately. The ISR figures are based on results from our interrupt latency benchmark and are coherent with the ones found in the literature such as [16]. The various configurable parameters were numerous enough to justify a friendly graphical interface, presented on Figure 3. We did not consider worth doing it for the other tests at this time.

Processor	Single iteration		ISR latency	
	Cycles	Duration (us)	Cycles	Duration (us)
333 Mhz Pentium II	106	0.318	295	0.885
233 Mhz Pentium II	104	446	NA	NA
120 Mhz Pentium	102	0.850	NA	NA

Table 4: Single iteration base time for the interrupt/preemption benchmark

Since the test pauses for relatively short periods of time, we were constrained to adjust the NT clock granularity to 1 ms. Otherwise, the effective pause time, which is a multiple of the clock granularity, would be rounded up to unacceptable values.

6.1. Influence of scheduling attributes

The win32 API maps the 32 real-time priorities to 4 classes (Normal, Idle, High, Real-time) of 7 priorities (**I**dle, **L**owest, **B**elow Normal, **N**ormal, **A**bove Normal, **H**ighest, Time **C**ritical). Priority 0 is reserved for a unique system idle thread. The standard default priority is 8 (Normal, Normal).

	Time Shared																Fixed Priority														
Class	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Idle		I	L	B	N	A	H									C																
Normal		I					L	B	N	A	H					C																
High		I										L	B	N	A	HC																
Real-time																	I						L	B	N	A	H					C

Table 5: Win32 scheduling classes

To confirm the effectiveness of NT's real-time scheduling we performed 4 distinct tests, each one over a 15 minute period:

1. The test running with standard scheduling attributes (priority level 8) on an idle system
2. The test running with a normal real-time scheduling attributes (priority level 24) on an idle system
3. Two simultaneous test instances running with normal real-time scheduling attributes (priority level 24).
4. One test instance running with time critical real-time scheduling attributes (Priority level 31) and a second test instance with normal real-time scheduling attributes (priority level 24).

Preemption length (µs)	Occurrences			
	Idle system		Background thread Pri 24	
	Pri 8	Pri 24	Pri 24	Pri 31
1 - 2	567	578	177	1612
2 - 4	410	648	291	161
4 - 8	358427	362081	282241	360316
8 - 16	54596	55695	42192	57376
16 - 32	7046	6253	5064	6487
32 - 64	1611	471	353	439
64 - 128	255	24	17	35
128 - 256	781	108	90	94
255 - 510	278			
510 - 1020	8			
1020 - 2040	8			
2040 - 4080	3			
4080 - 8192	500			
81912 - 16382				
16382 - 32764			5755	
Total occurrences	424490	425858	336180	426521
Max. Length	**5943**	**243**	**28950**	**246**
Active time	6'57"	7'3"	5'30"	6'59"
Intrs/sec	1033	1016	1016	1016

Table 6: Influence of scheduling attributes on preemption

First we clearly observe the effect of the real-time class scheduling, the maximum interrupt time drops from 6 ms to 243 µs. Second, it is confirmed that the scheduling policy is Round-Robin, with a quantum close to 30ms. We did not find any way to obtain or adjust the quantum value through the win32 API. Measurements on distinct platforms indicate that the quantum value appears to be a multiple of the standard NT clock resolution (15 ms in our case).

These figures already indicate that even with privileged scheduling attributes on an idle system, the preemption length for a pure cpu bounded task varies from 1 to 246 µs.

6.2. Work-load influence.

We now observe the predictability for the 9 types of workload. The configurable parameter values for the test are:

- 12 hour measurement period.
- 33 ms active period.
- 33 ms pause time.
- Normal real-time scheduling attributes (Priority level 24)

Preemption length (µs)	Occurrences								
	Idle	CPU	Make	Java	Disk	Net	Tty	CD	Floppy
1 - 2	5936	13781	51715	363403	290513	46328	562351	449676	17173
2 - 4	1148	181	613623	252435	327749	5341	2832	19644	3103
4 - 8	17431995	17371963	18462807	18426879	17141543	16114152	6123793	16541150	17649575
8 -16	2646904	2698110	3506763	3526674	5319637	6453197	207359597	3425003	4410545
16 - 32	292657	311801	261413	254619	903920	1195352	4130484	865993	349859
32 - 64	4859	5333	307417	190591	293509	1207523	29040225	150770	44221
64 - 128	724	876	20029	5497	4189	263354	24919	2693	1577
128 - 256	4767	4967	1197	634	2931	91166	2159	80	4563
256 - 512	13	4	3795	4575	2125	39422	2958	10	151
512 - 1024					1	4658			
1024 - 2048						4			
Total	20389023	20407016	23228759	23025307	564750	25420497	247249318	21455019	22480767
Max. length	294	286	409	384	547	1289	341	349	303
Active time (CPU)	5:34:28	5:35:1	5:30:14	5:34:26	5:28:17	5:31:13	4:42:22	5:35:12	5:38:13
Intrs/sec	1016	1015	1172	1147	1233	1279	14594	1067	1118

Table 7: Workload influence on preemption

The first observation is that intensive disk and networking activity significantly increase the maximum preemption durations, up to over **1 ms** values. Measurements on distinct hardware platforms have shown much larger values, such as 13 ms maximum preemption delays on an idle system, together with more frequent outliers especially for intense networking activity disk based activity. We do not have access to the driver sources and cannot provide precise explanations. For network drivers, in case of intense traffic it is often the case that the driver will process all packets buffered on the adapter at once, on a single hardware interrupt, and cause unexpected delays, proportional to the buffer size. [14] also explains that NDIS can cause long delays when checking for stalled miniport drivers. For disk drivers, copies of large memory buffers could certainly explain some delays. This may the case when the H/W is unable to DMA to the final location, such as for high memory addresses.

While developing this test we also observed that the use of floating point instructions and registers could cause unexpected latencies. We do not have any information on how floating point contexts are managed in NT, but it may be the case that NT uses lazy fsave/fresume policies. Our short sequence does not contain any floating point code.

7. Clock and timer services.

Real-time control relies heavily on clock and timer services. We provide 2 basic tests to measure the corresponding NT API and services. One to measure the sleepEx() API the other to measure the multi media timers. First, one needs to understand that a standard NT workstation is configured with a rather low timer resolution (10 or 15 ms). Services such as sleepEx() and the SetTimer() are directly dependent on this resolution and therefore unusable. One can change the timer resolution using the timeBeginPeriod() API down to a minimum resolution value of 1ms. The SleepEx() call is then more accurate, but the SetTimer() is unaffected. NT offers another timer (timeSetEvent()) for multimedia applications, which is dependent on the adjustable timer resolution. Such a timer may also be used for other critical applications, besides multi media ones.

7.1. SleepEx() API.

The test basically iterates over a SleepEx() API invocation. The effective sleep time is measured with the time stamp counter. The test is in fact more sophisticated to avoid stroboscopic effects. Simply iterating would always synchronize the next SleepEx() invocation with the last clock interrupt. So the test iterates CPU wise for a random period of time between 0 and the actual timer resolution (1ms in our case). We have performed the test first with the standard timer resolution (15 ms in our case) then with a 1ms resolution against the various workloads.

The configurable parameters for the SleepEx test were:

- 15 minute test period.
- 5 ms sleep time argument to SleepEx().
- Normal real-time scheduling attributes (priority level 24).

.

SleepEx() deviation (µs)	Occurrences									
	Std Clock	Clock granularity adjusted to 1ms								
	Idle	Idle	CPU	Make	Java	Disk	Net	Tty	CD	Floppy
-8192 : - 4096	1104									
-4096 : - 2048	7990									
-2048 : -1024	3899									
-1024 : - 512	1933									
-512 : - 256	979									
-256 : -128	477									
-128 : -64	226	5359	5839	4082	3919	4984	5337	3517	5370	5630
-64 : -32	120	4845	4878	4896	4849	4844	4866	4108	4872	4860
-32 : -16	59	2487	2467	2467	2498	2467	2449	2197	2496	2480
-16 : -8	28	1277	1303	1258	1303	1238	1268	1061	1244	1259
-8 : 0	25	1244	1266	1267	1225	1301	1240	1088	1281	1281
0 : 8	37	1282	1246	1247	1240	1228	1254	1122	1221	1272
8 : 16	15	1187	1220	1194	1228	1179	1227	1137	1192	1200
16 : 32	67	2407	2385	2388	2345	2467	2359	2146	2447	2379
32 : 64	116	4878	4903	4873	4895	4873	4874	4395	4840	4890
64 : 128	260	9663	9640	9635	9576	9638	9688	8925	9685	9629
128 : 256	482	19907	19897	19800	19814	19810	19854	18807	19899	19887
256 : 512	1003	39008	39018	38953	39027	39034	39020	38711	38993	39036
512 : 1024	2001	59012	58583	60237	60388	59428	59072	62243	59024	58799
1024 : 2048	4024	5	2	42	6	3	15	2117		4
2048 : 4096	7839									
4096: 8192	15635									
8192 : 16384	9282									
Total occurrences	57500	152561	152647	152339	152313	152494	152543	151574	152561	152606
Max. deviation	**10806**	**1068**	**1059**	**1506**	**1110**	**1099**	**1375**	**1448**	**907**	**1073**
Avg. deviation	3098	398	395	407	408	401	399	435	398	397

Table 8: SleepEx() timing deviations

The deviation is uniformly distributed within a [-0.1 ms to 1 ms] range (the index scale is logarithmic). This was expected, and means that the precision cannot be better than that of the timer resolution. The maximum deviation is prone to variations similarly to the preemption test. Measurements on distinct hardware with the same parameter configuration have shown deviation values as high as 4 ms.

We believe we can provide much better resolutions (down to µs values) using the Pentium on board timer/counter (local APIC) with a dedicated driver and specific APIs, in a similar way to that used to perform our interrupt latency measurements in section 8.

7.2. Multi Media timers

As opposed to a standard NT timer a multi media timer is implemented as a separate thread and fits much more naturally into a real-time context. Our test simply time-stamps every timer event with the time stamp counter value and compares the deltas with the timer programmed time value.

The configurable parameters for the test were:

- 15 minute test period.
- 5 ms period argument to timeSetEvent().
- Normal real-time scheduling attributes (priority level 24).

MMtimer drift (µs)	Occurrences								
	Idle	CPU	Make	Java	Disk	Net	Tty	CD	Floppy
-1024 : - 512			21			24	3253		
-512 : - 256			118	229	7	174	9246		
-256 : -128	1566	1327	9046	8205	13215	4946	23136	3816	2522
-128 : -64	156827	157069	148730	149700	145004	152752	105335	154565	155872
-64 : -32	4	2	254	33	158	204	1744	16	4
-32 : -16	1	2	43	2	5	44	818	1	2
-16 : -8			14			15	396		
-8 : 0			15		3	11	385	2	
0 : 8	2		18		1	6	392		
8 : 16			8		1	8	393		
16 : 32			14	3		20	802		
32 : 64			20	1	2	46	1562		
64 : 128			26	26	4	68	2789		
128 : 256			42	201		57	4339		
256 : 512			28			31	4414		
512 : 1024	21600	21600	21582	21600	21599	21576	19319	21600	21600
1024 : 2048			21		1	18	1677		
Total occurrences	180000	180000	180000	180000	180000	180000	180000	180000	18000
Max. drift	**898**	**887**	**1570**	**926**	**1090**	**1510**	**1514**	**929**	**955**
Avg. (absolute)	206	206	207	207	206	206	234	206	206

Table 9: Multi media timer drift

Since the timer events only occur at clock interrupt time, the deviation average is much smaller than that of a Sleep() like API, but the maximum deviation is comparable. It is interesting to note that the maximum deviation values are closely related to the number of interrupts per second, such as for the tty, net and java workloads (refer to table 3 for the workload characterizations) and underlines how dependent we are on other interrupt sources, whether or not of equal importance or priority as compared to clock services.

8. Interrupt latencies

We are interested in measuring the 3 basic latencies that characterize interrupt handling in windows NT (Figure 4). The ISR latency is the elapsed time between the real hardware event and the time at which the Interrupt Service Routine is invoked. The DPC latency is the elapsed time between the hardware event and the time the DPC routine is invoked. Note that it is not a requirement for drivers to use such DPC mechanisms. But generally minimum work is performed at ISR time and the bulk of the interrupt processing is done at DPC time. Finally the most important latency to the user is the time elapsed between the real H/W event and the time at which the thread handling the event is woken up. For a better analysis of the interrupt latencies, we will consider both absolute and relative latencies as depicted in Figure 4.

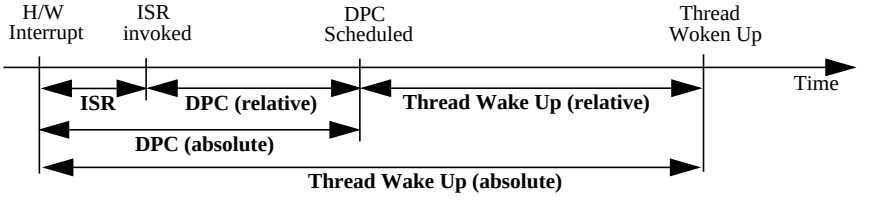

Figure 4: Interrupt latency categories

Most interrupt latency measurements are performed with the help of some dedicated HW board offering some high resolution timers. These measurements are therefore difficult to reproduce or to perform on new platforms. Beginning with the 735/90 and 815/100 models, and in all the P6 family, the Pentium processors include a local APIC with a programmable counter/timer. The specifications are publicly available in the Pentium System Programming Guide [19]. We have developed a dedicated kernel driver, associated with a user mode program to schedule interrupts and to time stamp the chronological events. The driver is designed and written in conformance with the Microsoft DDK [20] programming rules and we believe its architecture is close to that of typical windows NT drivers.

A user mode control program enables control (start and stop) of the APIC timer and adjustment of the timer frequency and mode (one shot or periodic). At ISR time the driver simply stores the TSC counter and the APIC counter values in a dedicated interrupt record location and then queues a DPC. At DPC time the driver also fetches the 2 counter values and adds them to the same record. The driver then returns the record structure to the user through an IoCompleRequest(). Once the thread is waken up it in turn fetches the counters and deducts the various latencies. The reason we use the APIC counter value in addition to the TSC value, is that the APIC counter keeps running once the timer interrupt has occurred and its relative value is used to compute the real hardware ISR latency, from the physical event time to the ISR invocation. We do not see other software alternatives for time stamping the real H/W event.

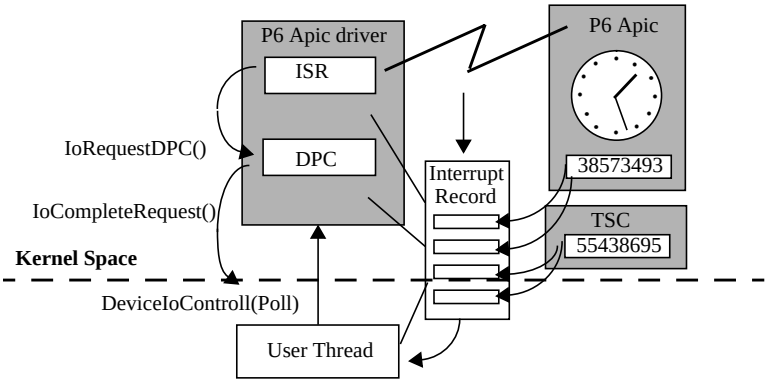

Figure 5: Local APIC driver architecture

We have performed the tests in the same conditions as for the previous tests, against the 9 types of workloads. We present the results as 3 distinct tables. The configurable parameters for the tests were:

- 15 minute test period.
- 10 ms interrupt periodicity.
- Time Critical Real Time Scheduling property. (priority level 31)
- IRQL 29.

The rather high IRQL value is quite un-realistic, since it is at a higher level than the clock IRQL (28), This is for historical reasons. However as we explained previously there is very little influence on interrupt processing ordering. In any case, a higher IRQL can only provide better figures, since the APIC interrupts are less often masked.

8.1. ISR latency.

ISR latency (µs)	Occurrences								
	Idle	CPU	Make	Java	Disk	Net	Tty	CD	Floppy
0 - 1		1							
1 - 2	85777	87828	3823	6	35190	74636	51707	61717	83853
2 - 4	4202	1890	47019	67810	48672	14935	24944	28108	5977
4 - 8	18	277	37956	22100	6095	420	177	172	164
8 -16	3	4	1056	84	41	8	240	3	5
16 - 32			2				470		1
32 - 64			12		1		943		
64 -128			12		1		1803		
128 - 256			22			1	3719		
256 - 512			52				5408		
512 - 1024			46				589		
Total samples	90000	90000	90000	90000	90000	90000	90000	90000	90000
Max. latency	**11**	**14**	**765**	**12**	**121**	**161**	**643**	**13**	**173**
Min. latency	1.21	0.975	1.21	1.89	1.08	1.15	1.05	1.06	1.03
Avg. latency	1.40	1.4	4.5	3.53	2.29	1.64	37.49	1.72	1.45

Table 10: ISR latencies

The relatively high maximum latencies (up to 765 µs) indicate that some kernel code remains masked against interrupts for relatively long periods of time. The real issue is that even the clock appears to be masked for such long periods (since our driver IRQL is higher than that of the clock). Observing the variation across the various workload it is certainly the case that this masked code ise located in interrupt handlers, but apparently not in the clock driver. It is impossible to state whether this misbehaving code is located inside ISR or DPC handlers.

8.2. DPC latency.

DPC latency (µs)		Occurrences								
		Idle	CPU	Make	Java	Disk	Net	Tty	CD	Floppy
2 - 4		1	32045							3
4 - 8		89588	57081	17714	20400	64284	85811	61691	86443	88315
8 - 16		396	851	67096	67138	24813	3765	9999	3299	1616
16 - 32		11	17	4353	2411	727	257	3015	183	54
32 - 64		1	2	566	43	169	128	2817	72	6
64 - 128		1	2	125	4	2	35	1824	3	2
128 - 256		2	2	37	4	5	3	3765		4
256 - 512				60			1	5807		
512 - 1024				49				1082		
Total samples		90000	90000	90000	90000	90000	90000	90000	90000	90000
Max. latency	**Absolute**	**192**	**203**	**771**	**245**	**180**	**343**	**692**	**84**	**248**
	Relative to ISR	**190**	**202**	**753**	**243**	**177**	**340**	**227**	**81**	**246**
Min. latency	Absolute	3.99	3.39	4.3	5.6	4.36	4.37	4.13	4.21	3.78
	Relative to ISR	2.58	2.16	2.62	3.32	2.67	2.65	2.58	2.67	2.42
Avg. latency	Absolute	5.19	4.68	11.48	9.87	7.44	5.75	48.11	6.09	5.31
	Relative to ISR	3.78	3.28	6.98	6.33	5.15	4.11	10.63	4.37	3.85

Table 11: DPC latencies

As a general observation, note that if the average absolute latency is the same as the sum of the average ISR latency and the average relative latency (such as 5.19 compared to 1.40 + 3.78 for the Idle workload), this is not the case for the minimum and maximum latencies (i.e 3.99 compared to 1.21 + 2.58 and 192 compared to 11 + 190). Indeed there is a very small probability that the worst case latencies (or minimum latencies) be observed for both ISR and DPC for the same sample. Comparing to the previous table, the worst case latencies confirm that some DPCs consume non negligible CPU resources, thus deferring other DPC regardless of IRQ levels. The lack of ordering for DPCs is visible from the Tty average latency. Otherwise, the low DPC latency of 3 to 4 us (relative to the ISR) is very reasonable.

8.3. Thread Wake-Up latency

Thread wake-up latency (µs)		Occurrences								
		Idle	CPU	Make	Java	Disk	Net	Tty	CD	Floppy
0 -32										
16 - 32		89888	89707	15895	9082	57022	85403	57236	82565	89136
32 - 64		106	289	68966	76113	32541	4233	18144	7380	815
64 - 128		3	1	4892	4798	432	289	3173	55	45
128 - 256		3	3	127	5	5	51	3690		3
256 - 512				58	2		20	6288		1
512 - 1024				62			4	1469		
Total samples		90000	90000	90000	90000	90000	90000	90000	90000	90000
Max. latency	**Absolute**	**244**	**222**	**807**	**284**	**203**	**663**	**715**	**105**	**268**
	Relative to DPC	**239**	**43**	**270**	**124**	**114**	**657**	**112**	**78**	**97**
Min. latency	Absolute	18.77	16.39	19.11	23.88	18.80	18.11	16.52	18.89	18.3
	Relative to DPC	13.71	12.77	13.98	17.49	13.66	12.95	12.03	13.68	13.67
Avg. latency	Absolute	20.69	20.04	43.50	44.91	30.81	23.74	70.88	24.35	21.08
	Relative to DPC	15.51	15.36	32.02	35.04	23.37	17.98	22.76	18.26	15.77

Table 12: Thread Wake Up latencies

Exactly the same observations as those made above for the DPC latencies regarding the comparison of the absolute and relative latencies apply here as well. For the Network activity bounded workload, there are large delays before the request thread is awoken. Using the SDK, DDK, and Resource kit we did not find any system thread (kernel or

user) running at level 31, this would indicate that there is heavy processing done at DPC time for networking.

If there is no way to fix NT so that it uses threads instead of DPCs for interrupt handling, rewriting or adapting drivers (provided that the sources or the H/W specifications are available) so that they use threads themselves would certainly improve predictability.

9. UDP/IP stack latencies

As we pointed out earlier, the distributed QOS dimension of the Quite project implies that windows NT networking predictability is critical for our project. Let us first state that a standard Ethernet based inter-connection is a real threat to determinism since the Ethernet protocol is by definition non predictable (unbounded re-emission delays). On the other hand, even if we had an ATM network and the associated NT driver, we are not interested in measuring how a particular driver performs but rather how the NT networking layer behaves. Hence we designed a pseudo virtual NDIS driver with the single purpose of measuring the network stack latency, without transferring data over a real network. We chose to emulate Ethernet at the physical layer and UDP as the upper level protocol since they require minimum code development in the driver. Also note that the UDP is a good candidate for Group Communication oriented protocols.

Since we desired the pseudo packet reception events to be unrelated to any software request, but rather mimic real packet reception we decided to re-use the local APIC driver to generate pseudo network packets. An other advantage is also the ability to precisely time-stamp the “pseudo” packet reception interrupt. So we have developed a NIC driver that interfaces with the NDIS library at the base of a UDP/IP stack on one side and interfaces the local APIC driver at the other end to simulate a network source. The local APIC driver as been extended to run in another mode where the ISR handler queues an alternate DPC, belonging to the pseudo NIC driver.

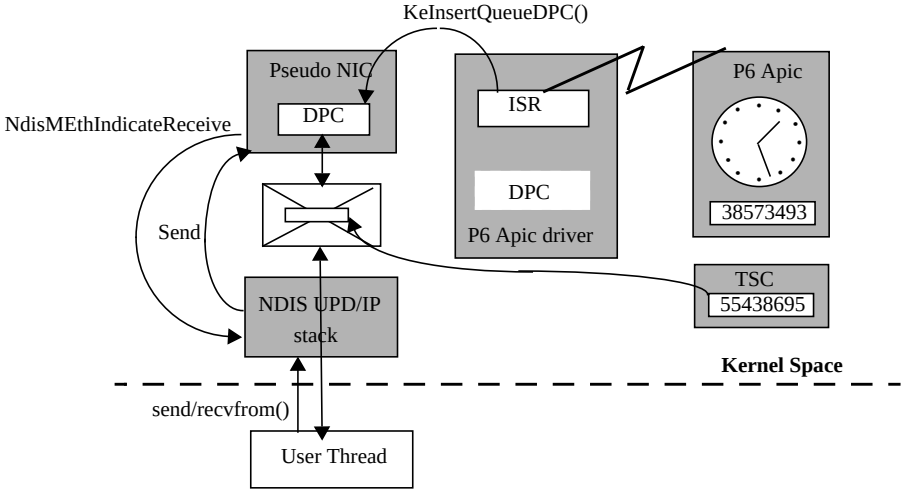

Figure 6: NIC pseudo driver architecture

Since the pseudo NIC driver is not connected to any real network, the driver must be prepared to handle some well known packets so that the windows NT networking services and registry be properly configure and so that no network exception events increase the noise. The driver handles a subset of the ARP, ICMP, and NET_BIOS protocol packets. The ICMP is fairly useful for test purpose. The driver also handles the UDP based packet format that our test uses to measure the latency.

The measurement sequence for sending and receiving packets is asymmetric. To measure the receive side, the test program controls the APIC driver to run in network performance measurement mode (i.e. use an alternate DPC belonging to the pseudo NIC driver), sets the periodicity and then invokes recvfrom() iteratively. Each time it receives a new packet, the test fetches the current value of the Time Stamp Counter and subtracts the TSC value register in the packet. To measure the send side, the test itself timestamps a packet that it sends via the socket send() API, the packet is forwarded through the NDIS stack to the pseudo NIC, which immediately time stamps the packet. The driver must return a packet with the time stamp counter value, but this step is not accounted in the measurement. Also note that the send side does not require any support from the local APIC. The send side

measurement requires two UDP messages, whereas the receive side only requires one.

The test only requires 2 long words (32 bit) to be stored in the network packets, one for the Time Stamp Counter the other one as a packet number. So the overall network packet size is 54 bytes.

9.1. UDP/IP Receive latency

The configurable parameter values are:

- 15 minutes test
- Time Critical Real Time scheduling attributes (Level 31)
- 10 ms interval between packets.

UDP/IP Receive latency (µs)	Occurrences								
	Idle	CPU	Make	Java	Disk	Net	Tty	CD	Floppy
32 - 64	89206	89442	22285	12343	58412	86892	55675	88116	89482
64 - 128	76	534	63947	74378	30821	1493	33194	1189	482
128 - 256	2	23	3704	3956	93	137	410	25	36
256 - 512	2	1	55	4	4	93	7		
512 - 1024			8			12			
1024 -2048			1						
Total samples	89286	9000	90000	90681	89330	88627	89286	89330	90000
Max. latency	**275**	**265**	**1033**	**381**	**314**	**775**	**352**	**181**	**225**
Avg. latency	42.69	42.39	80.25	84.24	58.73	47.11	62.84	47.00	43.63
Elapsed time	14’53”	15’00”	15’00”	15’07”	14’53”	14’46”	14’52”	14’53”	15’00”

Table 13: UDP/IP Receive latencies

The latencies do not appear worse than the preemption latencies for a simple CPU program. We would need to run the test for longer period of times and a higher periodicity for increased confidence. Still the results are encouraging.

9.2. UDP/IP Send latency

The configurable parameter values are:

- 15 minutes test
- Time Critical Real Time scheduling attributes (Level 31)
- 10 ms interval between packets.

UDP/IP Send latency (µs)	Occurrences								
	Idle	CPU	Make	Java	Disk	Net	Tty	CD	Floppy
16 - 32	77180	76870	1386		24707	46976	26087	49275	74068
32 - 64	6877	7190	44671	41183	47798	35941	53749	34503	9200
64 - 128	16	13	35715	42206	11543	489	3851	291	155
128 - 256			1664	724	24	17	1	4	27
256 - 512			14		1	27			
Total samples	84073	84073	83450	84113	84073	83450	83488	84073	83450
Max. latency	**84**	**77**	**288**	**187**	**268**	**462**	**149**	**135**	**172**
Avg latency	27	28	65	69	45	31	41	33	28
Elapsed time	15’3”	15’3”	14’56”	15’3”	15’3”	14’56”	14’57”	15’3”	14’56”

Table 14: UDP/IP Send latencies

As compared to the receive case figures, the latencies are smaller, as expected. The send test is not interrupt driven, it simply performs a socket() call and proceeds in the kernel, for a fairly small duration. It is then more immune to interrupts or preemptions than for the receive case.

10. Using NT real-time extensions.

We have investigated 3 commercial Real Time extensions to NT. Our study was limited to reading the various company provided documentation as well as some performance studies such as [2] and [4]. We did not experiment

with the extensions, neither did we perform any measurements. The 3 extensions are:

- HyperKernel from Imagination Systems Incorporated [5].
- INtime kernel from Radisys Corporation [6].
- RTX Real-Time Subsystem from VenturCom [7].

The three extensions are not architectured nor implemented identically. HyperKernel and INtime are similar in the sense that the extension is implemented as an aside protected kernel running with higher privilege than the NT kernel. RTX is implemented as an NT subsystem inside which RT threads also have precedence over any other NT activity. The three extensions offer specific RT APIs plus some interfaces to communicate between NT and RT threads. We will not describe other differences, but rather emphasize the key architectural elements.

As depicted in Figure 7, functionally the whole NT system acts as an RT thread of the RT kernel extension, but with the least privileges (lowest priority). Note that the HyperKernel has an optional anti-starving mechanism for NT, but at the cost of higher worst case timings. The RT extensions offer FIFO scheduling, more priority levels, synchronization mechanisms with priority inversion handling. Dedicated drivers for real-time control devices would be implemented as RT threads. A subset of the WIn32API is available to the RT threads, with predictable service times. The NT threads and the RT threads can communicate using messages or shared memory.

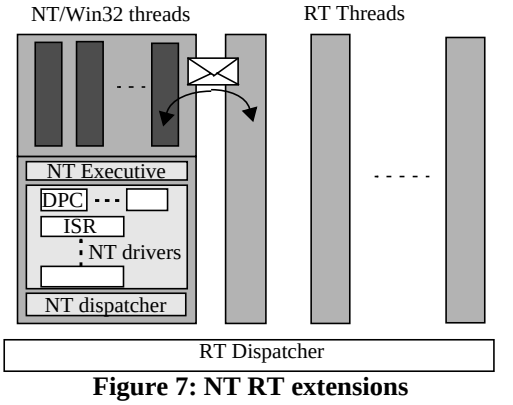

Figure 7: NT RT extensions functional architecture

Typically, a real-time control application is written in such a way that the time critical components run as RT threads on top of limited but predictable kernel services. On Figure 8, the robot arm is controlled by an RT thread (#1). Whichever activity occurs on the windows NT system, the RT thread would preempt it at any time, including when processing interrupts such as disk transfer ones. The other components, less critical, may be implemented as NT threads and access the full Win32 API. In our example, NT thread #2 acquires data from RT thread #1 and logs it onto disk. Threads #1 and #2 communicate with messages or shared memory through an API specific to the extension.

Communication and synchronization between the application time critical threads (#1) and non critical threads (#2) must be carefully designed to prevent priority inversion like symptoms. For instance, if thread #1 and thread #2 share some buffer or mail box to exchange data, the buffer may overflow while waiting for disk IOs to complete, and it should not block thread#1. Despite the fact that the application designer may have ensured that the overall system could sustain the highest data throughput, NT is unfortunately not deterministic.

Note that this design phase, where one must prioritize the tasks, choose the appropriate synchronization and analyze schedulability is also applicable to a pure NT solution, where the time critical components would be written such that they would rely as little as possible on un-predictable Win32 services. Still we have demonstrated that even pure CPU tasks are not immune to unexpected delays on windows NT and this is where the real time extensions are much more appropriate. So clearly, RT extensions for NT can drastically improve predictability, certainly with sub-milliseconds worst case latencies if the application is adequately architectured.

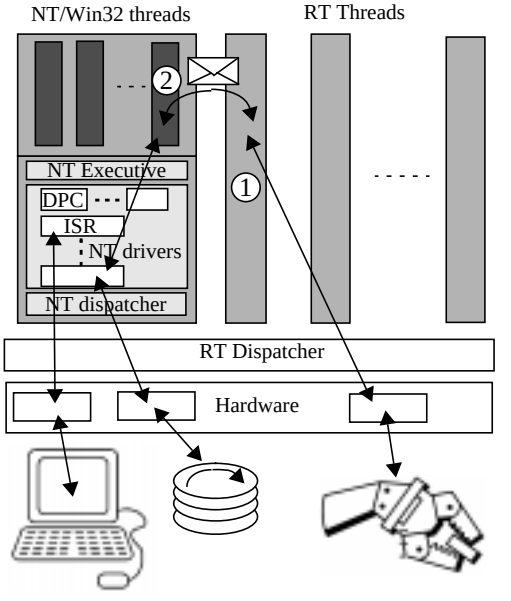

Figure 8: Typical real-time control with RT extensions

Let us now analyze how this applies to the Quite environment and what the impacts are. As compared to a typical real-time control application the primary focus of Quite is Distributed QoS. This means that not only the individual nodes may control dedicated hardware such as radars, but it is required that communication and failure detection with participating nodes be predictable. The same way the robot arm is controlled by an NT thread, network communication must be implemented on the NT extension. It is not conceivable to rely on the NT side device drivers and protocol stacks since we would lower our determinism to that of NT, thus completely loosing the benefits of the real-time extension. The consequences are that 1) the network device drivers must be written especially for the extension, 2) the network stack protocols (ATM, ARP, IP, UDP ...) must be ported to the extension as well. In the case of a typical real-time control application, the H/W controllers are dedicated, often not supported by NT, so it is required that a driver be developed anyway. In our case, we are not able to re-use the NT network drivers, even though most network cards are supported by NT. Adapting the standard IP protocols may require significant engineering effort to enforce per session QoS. We did not find any references to IP support in the documentation of he extensions, nor references to supported drivers for ATM or predictable network interfaces.

Finally, we must address the issue of sharing the communication link between NT and the Real-time extension. It is highly probable that the NT side application component will require access to the network. It appears unrealistic if not impossible to share the network interface driver between NT and the extension. We would once again lower our determinism to that of NT. Instead we can think of two deterministic solutions.

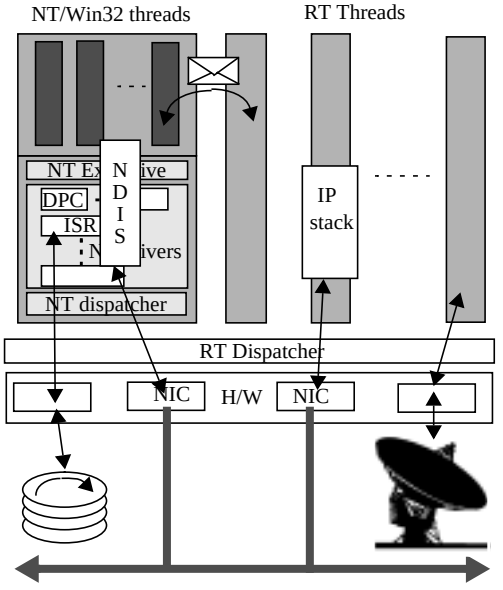

Figure 9: Dual NIC approach

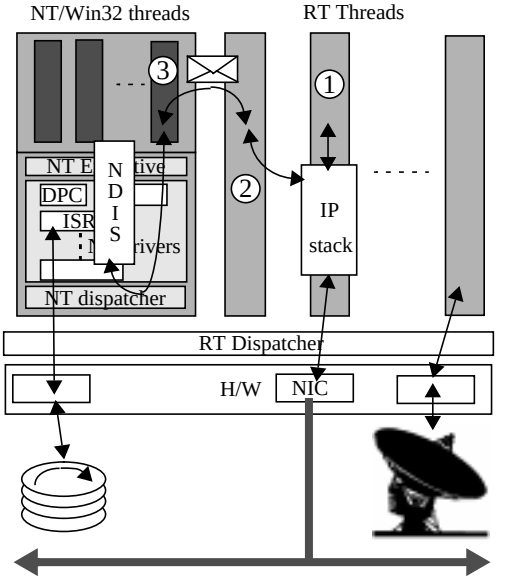

Figure 10: Shared NIC approach

1. The first approach (Figure 9) consists in driving two network interfaces, one handled by NT, the other by the Real-time extension.
   Advantages: - It does not require additional software (but it still requires a NIC driver and protocol stack on the extension side.
   - It allows eventually to drive two networks, with distinct traffic types and predictability.
   Drawbacks: - The requirement for a second network interface.
   - Some administration complexity since the host will have two addresses.

2. The other approach (Figure 10) would be to drive the single network interface on the extension side. All packets would be at first filtered by a high priority thread (#1). Packets that are not relevant to the extension side components would be immediately handed off to a low priority thread (#2) and later on transmitted to an NT thread (#3). This thread would act as a software NIC driver and communicate with an NDIS stack.
   Advantages: - It does not require an additional interface
   - The host is configured with a single address.
   Drawbacks: - It requires development of a packet filter and a forwarder on the Real-time extension side, and a pseudo software NIC driver on the NT side.
   - The software must be carefully designed so that the handling of non critical network packets does not induce unpredictable latencies on the critical tasks.

11. Conclusions

We have summarized the worst case figures for each individual test in table 15.

Test		Worst case latency or deviation (µs)								
		Idle	CPU	Make	Java	Disk	Net	Tty	CD	Floppy
Periodic CPU Preemptions		294	286	409	384	547	1289	341	349	303
Timer services	Sleep	1068	1059	1506	1110	1099	1375	1448	907	1073
	Timer	898	887	1570	926	1090	1510	1514	929	955
Interrupts	ISR	11	14	765	12	121	161	643	13	173
	DPC	192	203	771	245	180	343	692	84	248
	Thread Wakeup	244	222	807	284	203	663	715	105	268
UDP Stack	receive	275	265	1033	381	314	775	352	181	225
	send	84	77	288	187	268	462	149	135	172

Table 15: Worst case latency and deviation figures summary

Among the measurements, the make based workload causes some of the worst latency figures. We must admit that we cannot really explain why, since interrupt wise, the disk based workload does not show similar figures, and the compiler input and output files are local. The disk transfer sizes are also comparable in both workloads (table 3). Running all tests for longer periods of time would certainly give better confidence.

In any case, **for this particular platform**, the figures clearly indicate that it is unwise to rely on deviations **inferior to 2 ms**. This is relatively high. The study also demonstrates that without any additional in-depth information (i.e. source code), it is extremely hard to locate precisely the components or drivers that cause the worst case latencies, and whether or not the empirical measurements captured, or have the ability to capture, the worst case figures
More generally, the **small number of priority levels** and the **absence of FIFO** scheduling may be a strong obstacle to building real-time applications.

It is important to realize that there is no way of proving formally what the worst case figures are for NT, we simply offer some standard tests that can be used to perform measurements empirically. Rewriting or adapting some NT drivers, such as the disk or network ones, to use threads instead of DPCs would certainly decrease the worst case latencies. New measurements would then be required.

In the context of the Quite project, the use of real-time extensions would undoubtedly enhance the predictability but at the cost of at least **developing dedicated network drivers** (ATM like) and a full **protocol stack** specific to the extension.

12. References

[1] Real-Time Systems With Microsoft Windows NT. Microsoft. - http://www.microsoft.com/embedded/winnt.htm

[2] Windows NT Real-Time Extensions better or worse? - http://shop.realinter.net/rtshop/registered/winntr-text.asp

[3] Can Windows NT 4.0 be used as a RTOS - http://shop.realinter.net/rtshop/registered/winntasrtos.asp

[4] Hard Real-time Extensions of Windows NT[tm] Evaluation Report - http://www.arcweb.com/omac/Docs&NRs/ntrtrpt2.pdf

[5] Hyperkernel from Imagination System Inc. http://www.hyperkernel.com/paper1.html.

[6] InTime form RadiSys. http://www.radisys.com/products/intime/explained.html

[7] The RTX Real-Time Subsystem for Windows NT. http://www.vci.com/products/real_time_products/rtx/rtx_index.html

[8] Real-Time magazine- http://www.realtime-info.be

[9] Inside NT's Interrupt Handling. Windows NT magazine. Mark Russinovich - http://www.winntmag.com/magazine/Article.cfm?IssueID=25&ArticleID=298

[10] Windows NT Architecture, Part 1 - Part 2. Windows NT magazine. Mark Russinovich - http://www.winnt-

mag.com/magazine/Article.cfm?IssueID=29&ArticleID=2984

[11] Inside NT's Interrupt Handling. Windows NT magazine. Mark Russinovich - http://www.winntmag.com/magazine/Article.cfm?IssueID=25&ArticleID=298

[12] Advanced DPCs - Mark Russinovich - http://www.sysinternals.com/dpc.htm

[13] Inside Windows NT. Helen Custer ISBN 1-55615-481-X

[14] Issues in Using Commodity Operating Systems for Time-Dependent Tasks: Experiences from a Study of Windows NT. Michael B. Jones, John Regher. Eighth International Workshop on Network and OPerating Systems Support for Digital Audio and Video (NOSSDAV 98), Cambridge, England, pages 107-110, JUly 1998.

[15] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority INheritance Protocols: An Approach to Real-Time Synchronization", IEEE Transactions on Computers, Vol. 39, No. 9, 1990.

[16] QNX Neutrino System architecture Guide - http://www.qnx.com/literature/nto_sysarch/kernel3.html

[17] Windows NT as Real-Time OS? - http://www.realtime.info.be/encyc/magazine/97q2/winntasrtos.htm

[18] Windows NT for Soft Real-time Control. Separating the fact from the fiction http://www.mcdonald.com/articles/rsi/index.html

[19] Pentium System Programming Guide - ftp://download.intel.com/design/pentium/MANUALS/24319201.pdf

[20] Microsoft DDK - http://premium.microsoft.com/msdn/library/?FinishURL=/msdn/library/

Appendix A - Time stamp counter macros

The following macros enable access to the Pentium TSC register with minimum overhead, from both user mode threads and from any kernel module. The syntax is presently dependent on Microsoft's compiler (more precisely SDK or DDK in our case) but should be easily adaptable to other compilers.

```
#define rdtsc __asm _emit 0x0f __asm _emit 0x31

/* 32 bit variant */

#pragma warning( disable : 4035 )
__inline int read_timestamp()
{
  __asm {
    mov edx, edx; Just so that edx be saved or unused in upper block */
    rdtsc;
  }
}

#pragma warning( default : 4035 )

/* 64 bit  variant */
#pragma warning( disable : 4035 )
__inline LONGLONG read_Xtimestamp()
{
  __asm {
    rdtsc;
  }
}

#pragma warning( default : 4035 )
```

Appendix B - interrupt/preemption benchmark filtering method

The screen capture (Figure 11) of a pop up window for the preemption test illustrates how we extract the preemption latency figures. The goal of this test is to measure the variations for interrupt and preemption durations. As explained in section 6, we measure the duration of a short fixed sequence of code (fixed number of cycles). The data sampling and gathering must be part of this code sequence, otherwise the test would potentially miss interrupts while gathering each figure. So whether or not the code sequence is interrupted, its duration (less the base reference time) is recorded. This explains the first row of figures on the screen capture, accounting for 99.96% of the occurrences with an average null duration deviation (offset as compared to the minimum reference time), representing un-interrupted sequences.

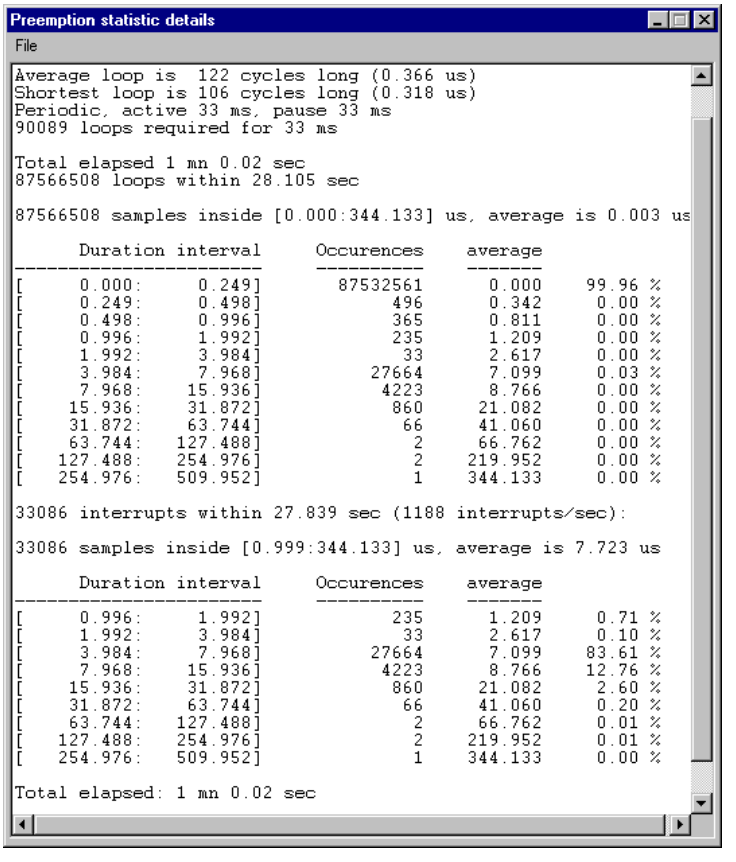

Figure 11: Preemption test text mode output

We consider that the sequences that are really interrupted are the ones for which the duration variation (offset with reference) is at least 4 times as large as the reference duration (4 x 0.313 = 1.252 µs). Note that using a lower ratio such as 2 or even 1 would not significantly change the results. We empirically adjusted the ratio to 4 to match the interrupt/sec. ratio to that of the NT performance monitor and believe that the very small deviations are related to translation or memory cache misses. This ratio is adjustable at test start-up.