Measuring the Latency Time of Real-Time Unix-like Operating Systems

Newton Faller

TR-92-037

June 1992

Abstract

With the advent of continuous-media applications, real-time operating systems, once confined to process control and other specialized applications, are coming to the desktop. The popularity of UNIX made this operating system the first choice for use with such real-time desktop applications. However, since UNIX kernel does not provide real-time responsiveness, some software developers have been trying to adapt it to respond to this new requirements, while others have been proposing its total redesign. Though the evaluation of the performance of a real-time operating system depends on many factors, a predictable small latency time in responding to external events is always essential. In this paper, after a discussion about the probable sources of latency, it is presented a method for collecting information about context-switching and interrupt-acknowledge times in UNIX-like operating systems without requiring external measuring tools. It is also proposed, a form of presentation of these data aimed at facilitating the comparison with previously collected data obtained from the same or from other systems. The paper is illustrated with actual results obtained by the application of the method to TROPIX, a real-time UNIX-like operating system, running on a Motorola 68010-based computer. The impact of kernel preemption and some practical measurement interference considerations due to dynamic memory refresh, DMA operation and disk multiblock access are also discussed.

This research was supported by CNPq - Conselho Nacional de Desenvolvimento Científico e Tecnológico (Brasil) and also by the National Science Foundation and the Defense Advanced Research Projects Agency (DARPA) under Cooperative Agreement NCR-8919038 with the Corporation for National Research Initiatives, by AT&T Bell Laboratories, Hitachi, Ltd., Hitachi America, Ltd., Pacific Bell, the University of California under a MICRO grant, and the International Computer Science Institute. The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing official policies, either expressed or implied, of the U.S. Government or any of the sponsoring organizations.

Measuring the Latency Time of Real-Time Unix-like Operating Systems

Newton Faller

International Computer Science Institute
Berkeley, CA, U.S.A.
faller@icsi.berkeley.edu

Universidade Federal do Rio de Janeiro
Rio de Janeiro, RJ, Brazil
faller@ufrj.bitnet or faller@rocinha.nce.ufrj.br

Abstract

With the advent of continuous-media applications, real-time operating systems, once confined to process control and other specialized applications, are coming to the desktop. The popularity of UNIX made this operating system the first choice for use with such real-time desktop applications. However, since UNIX kernel does not provide real-time responsiveness, some software developers have been trying to adapt it to respond to this new requirements, while others have been proposing its total redesign. Though the evaluation of the performance of a real-time operating system depends on many factors, a predictable small latency time in responding to external events is always essential. In this paper, after a discussion about the probable sources of latency, it is presented a method for collecting information about context-switching and interrupt-acknowledge times in UNIX-like operating systems without requiring external measuring tools. It is also proposed, a form of presentation of these data aimed at facilitating the comparison with previously collected data obtained from the same or from other systems. The paper is illustrated with actual results obtained by the application of the method to TROPIX, a real-time UNIX-like operating system, running on a Motorola 68010-based computer. The impact of kernel preemption and some practical measurement interference considerations due to dynamic memory refresh, DMA operation and disk multiblock access are also discussed.

1 - Introduction

A real-time operating system may be defined as a system such that, for a defined maximum workload, it is always capable of acknowledging a specific set of asynchronous external events and execute all the processing required to respond to them within a finite and predictable amount of time compatible with the application. Operating system latency time is the real elapsed time between the occurrence of a specific external event and the appropriate system response to it.

Complex computer-based real-time systems have been controlled by real-time operating systems. These systems, once confined to process control and other special applications, are replacing time-sharing operating systems due to the stringent requirements of continuous-media applications. In fact, in the near future, it is expect that they will become commonplace in desktop workstations.

Real-time systems have been divided in hard and soft [StRa88]. They basically differ in the way scheduling is done. While in hard real-time systems a set of tasks with their deadlines are specified, in soft real-time systems the highest priority task should be executed as fast as possible. In this paper we will be concerned with soft real-time systems only.

With the UNIX system being the most popular operating system for workstations, it is easy to understand that many software developers have been trying to fit UNIX to run appropriately with real-time applications. This has been done through adaptation of its kernel, for instance [KhSZ92] [Fual91], or even, sometimes, through the design of a new kernel [FaSa90].

There are many ways of estimating the performance of a real-time operating systems. In [Fual91], for instance, four metrics are described: the Rhealstone metric, the Process Dispatch Latency Time, the Tri-Dimensional Measure, and the Real/Stone Benchmark. In all of them, latency time has always been one of the main factors.

In this paper we suggest a method for obtaining interrupt response and context switching latency times in Unix-like operating systems requiring no external measuring tools. It is assumed, however, that the hardware has a real-time clock. The operating system is not required to be specifically designed for real-time applications but it should have some similarity to the UNIX system. A description of the procedures to be executed and the software instrumentation required for the measurements to be obtained are also described.

As an example, the method was applied to TROPIX, a real-time Unix-like operating system [FaSa90], which source code was readily available to us. The hardware used was a 10MHz one-wait-state Motorola 68010-based computer (EBC-32010) manufactured by EBC, a Brazilian computer company. Though somewhat outdated, it was chosen because it was the only available with a ported version of TROPIX, we already had all the hardware information required [EBCto87][EBCde87], and it fitted very well the purpose of illustrating the proposed methodology.

The data obtained are presented in a special logarithmic-exponential percentile graph. This type of graph permits an easy visual comparison between results obtained from the same operating system when running with different generation parameters or under different pre-defined workloads, or from other operating systems when submitted to comparable workloads. This method can be used, and actually has been used, as a gauge to evaluate the improvement of system responsiveness as modifications in the TROPIX kernel are performed.

2 - The Hardware Environment

Real-time performance evaluation of operating systems can be a very elusive task if one does not have a very thorough understanding of the hardware where the operating system is running. Dynamic memory refresh, direct memory access controllers (DMA), caches in disk controllers and in the central processor, to mention just a few examples, may cause severe unexpected interference in the measured data to a point that might turn them difficult to interpret. Therefore, when using the methodology presented here, though not essential, it is rather important to have available for consulting the target computer system manuals that explain its theory of operation [EBCto87], contain its electrical diagrams [EBCed87], and describe its central processing unit with timing [MOTpr84], peripheral circuits [MOTfr90] and peripheral devices.

The methodology described in this paper, was applied, as an example, to an operating system running on a EBC-32010, a 16-bit bus computer based on a 10 MHz Motorola MC68010 microprocessor. This is a simple computer with main memory and all peripheral circuits sharing a single common bus.

Its main memory has 3 Mbytes made up of dynamic memory chips which words (16 bits) can be accessed by the microprocessor with one clock-period wait state. Since the simpler instructions take four clock periods plus one memory access to execute, and a typical instruction might require eight periods plus two memory accesses, the EBC-32010, to allow for further comparisons, can be considered an approximately 1 MIPS machine.

Direct memory access is provided through the four channels of a MC68450 Motorola DMA controller which were programmed to function in the cycle-steal mode. Each channel can also be programmed to transfer data from/to peripherals either one block at a time or multiple blocks through chaining the commands. The TROPIX operating system allows choosing the method of transfer, which has a reasonable impact in latency time.

Dynamic memory refresh is performed by the dummy continuous chained access of a block of memory through one of the DMA channels. Independent measurements showed that the refresh operation takes 11 of every 128 clock periods, which accounted for the consumption of 8.6% of the bus bandwidth. The data obtained during the measurement sessions were not compensated for this wasted refresh time and, therefore, the results presented include this overhead.

Hard disks are connected through a slave (non-disconnect) 8-bit SCSI (Small Computer Standard Interface) interface. Floppy disks have an independent controller but were inactive during the measurement sessions. Hard disks and floppy disks transfer data

through different DMA channels. The hard disk connected to the system was a 85 Mbyte Seagate 296N (3600 RPM, 820 cylinders, 6 surfaces, 34 sectors of 512 bytes per track).

Serial I/O is provided by four MC68681 DUART's (Dual Asynchronous Receiver/Transmitter) making possible to have up to eight physical serial lines connected to the EBC-32010. These DUART's were programmed for 8-bit characters, no parity, one start bit and one stop bit during the measurement sessions.

A parallel port is also present in the EBC-32010 but it was inactive during the measurement sessions.

Clock interrupt is assigned to the highest maskable priority, which in the MC680X0 family is 6. An interupt of this level is generated every 10 milliseconds by a MC68230 timer. This timer actually generates this interrupt when one of its previously loaded decreasing internal counter reaches the value zero. At this point, the counter is automatically reloaded with a pre-set value which provides the appropriate generation of the next interrupt. In the EBC-32010, the oscilator that drives the processor is the same that drives the counter and the circuit was designed in a way that this counter, with some caution, despite its ripple carry characteristics, can be read by software any time. The information contained in this counter together with some additional time information stored in the kernel of the operating system, makes it possible to implement a 64-bit real-time clock with a resolution of 3.2 microseconds. This resolution was used in all measurement sessions.

3 - The Operating System Environment

The measurement methodology which is described in the following section, was applied to the TROPIX operating system. TROPIX is a fully-preemptible real-time UNIX-like operating system [FaSa90] which design was made possible through the experience gained in the design of PLURIX, a multiprocessing UNIX-like operating system [FaSa89][Faal84].

In the user level, TROPIX bears a reasonable similarity to the UNIX operating system. Processes are created through fork-execs, I/O is always treated as a sequence of bytes and is performed through open-read-write-close primitives, signals can be sent to processes, there is a kernel process zero (*swapper/pager*), the *init* process is the common ancestor of all other user processes, etc.

Internally, TROPIX kernel structure [FaSa90] is quite different from UNIX's [Bach86]. TROPIX has a fully-preemptible kernel and many specialized system calls to manipulate and coordinate the execution of real-time processes. Real-time processes coexist with their time-sharing counterparts but they can run at higher priorities and have many other privileges.

Besides its *swapper/pager*, TROPIX kernel standard processes include a unique *dispatcher* process per processor. When running in a multiprocessing environment, this scheme greatly facilitate the implementation of different scheduling strategies to be followed by different processors. Fine-grain parallel processing within executing processes is also possible since TROPIX implements threads at the supervisor level.

For the understanding of the measuring methodology described in the next section, however, there are just a few characteristics present in the TROPIX operating system, and common to most UNIX-like systems, which are important to pinpoint. These

characteristics are described in the following paragraphs.

The TROPIX operating system process zero is executed once a second and starts some housekeeping operations related to main memory allocation. This process runs at the highest priority of all time-sharing processes. Real-time processes, however, run, in general, at higher priorities.

In the TROPIX operating system, besides the data structures sizes which can be dynamically allocated at boot time, there are a few parameters that can be modified at any time through special system-calls. From those, the ones, which are important for our measurement sessions, are the preemption control and the multiblock control parameters. When preemption is turned off, context switching to a new process, due to the occurence of an external event, is not executed immediately upon the occurence of the corresponding interrupt if the process is executing in supervisor mode. It is delayed until the current process willingly relinquishes the control of the processor. When preemption is on, context switching may occur immediately following the occurence of an external event, no matter in which mode the current process is presently executing. Actually, there are a very few points where preemption is always inhibited. One of them, for instance, is when the current process is already switching its context.

In the TROPIX operating system, the size of disks blocks is always 512 bytes. When multiblock is turned off, blocks are read from disk one at a time. This means that the DMA and the device controller are programmed and interrupt processing takes place each time a block is read. When multiblock is turned on, the DMA and the device controller are programmed once and many blocks can be read in one single operation. When no limiting parameter is given, an entire file can be read with a single DMA-controller programming operation. This procedure substantially improves I/O performance when requested disk blocks are physically contiguous in the media and a substantial amount of data in a disk track can be read in a single disk revolution. Multiblock transfers, however, may monopolize the bus for an unreasonable period of time and may become one of the main sources of latency. Anyway, the multiblock control parameter has influence just in block reads since block writes, except for swapping or paging, are always performed as single block operations.

4 - The Sources of Latency

In this paper, latency is defined as the delay presented by an operating system to start a processing triggered by an external event. It is the time it takes between the occurence of an external event and some action, related to this event, that the computer system is supposed to execute. Though this action is ultimately a physical action, it has been interpreted in different levels of abstraction and are treated with different priorities. Therefore, initially, it is important to understand the three levels of priority with which a computer system gives attention to external events: DMA operations, interrupt operations and process operations.

When attention to external events is extremely critical, it is essential to use DMA operations. The priority of DMA operations is the highest among all other above mentioned operations in a computer system. In addition, each channel of each DMA controller can be further prioritized to garantee that the processing required by the occurence of an external event will not be lost. The latency of a high-priority DMA operation is determined by the bus arbitration protocol, which is executed in hardware, and by the

way the bus is used (reservation and release). When multiple equally important events can occur basically at the same time and must be treated without delay, it is reasonable to consider the use of a multibus architecture with multiport memories. This is actually the typical configuration of hardware designed for critical real-time applications.

One may argue that the processing of a DMA controller is too simple (read/write) to be considered as a task on a computer system. However, between a simple read/write operation that a DMA controller can perform and the complex data manipulation a full processor can execute, there is a full range of operations that a specialized processor may take advantage of using the shortest latency time a computer system can provide through direct access to its main memory. Actually, it has been reported that some high speed communication boards when connected to workstations do not rely on interrupts or user processes to perform some of their critical tasks. They manipulate data structures directly on workstation memories, for example.

When attention to external events is not extremely critical, one can rely on interrupt routines to process them. Like DMA operations, interrupt operations may be further prioritized. The latency of the highest level interrupt operation in a computer system may be influenced by three factors: instruction time execution, DMA operations, and software masking.

Since, by definition, interrupt operations can only start after the current instruction is fully executed, instructions that require a long execution time may cause an unreasonable delay to interrupt operations. There is nothing an operating system can do about this. Main processors with very long execution time instructions may save intermediate instruction status (similar to what is done by some processors when a page fault occurs in the middle of a instruction execution) and allow an interrupt operation be executed before the current instruction is completed. This procedure, however, is costly and is rarely used.

DMA operations have a higher priority over interrupt operations because DMA controllers have a higher priority in holding the memory bus than main processors. There are parameters in DMA controllers to avoid the starvation of main processors unable to hold the bus for a long period of time. These parameters specify, in general, a duty cycle limit for the DMA controller and should be set accordingly by the operating system.

Though not being possible to totally avoid DMA controller interference, there are some ways in hardware to minimize it. The use of multiport memories with multiple buses is one, the use of processor caches is another. The main objective here is, upon the occurrence of a high priority interrupt, to avoid as much as possible the contention for some bus among the main processor, which is going to handle the interrupt, and DMA controllers currently in operation.

Software masking is the main cause of delay of interrupt operations caused by the operating system. The operating system manipulates several data structures which, for safety, must be left in a consistent state before executing an interrupt operation. The operating system raises the interrupt level of its processor to the highest maskable level allowed not permitting, therefore, that interrupts occur during the period the level was raised. We should not forget that in a multiprocessing environment just raising the interrupt level of a processor is not enough to guarantee exclusive use of resources. This concern, however, is not important for our measuring methodology.

Finally, process operation is executed by a processor after the operating system decides that this specific process should run because the condition which it was waiting for does not exist anymore. Like DMA operations and interrupt operations, process operations can be prioritized. The latency time for a high-priority process operation to be initiated is influenced by the current DMA and interrupt operations which are taking place at this specific moment. We should not forget that as long as these operations are active the main processor cannot execute the code which ultimately will lead to the context switching from the current to the desired process.

One of the operating system functions which may turn out to be a source of long latency times for interrupt acknowledge and context switching because of the potential high DMA activity is paging and/or swapping. Certainly, real-time process are always resident in the main memory during their entire execution time and do not suffer delays in their execution due to missing pages. However, their latency times may be severely affected by paging/swapping activities of concurrently executing time-sharing processes. Many real-time operating systems just disable their paging/swapping capabilities (if they have them at all) when running critical real-time applications.

When there are no DMA or interrupt operations active, the context switching should be performed as soon as possible. The importance of a preemptible kernel becomes evident if one is concerned with low latency context switching time. But still, there are points in the kernel code where it cannot be preempted. During a context switching, for instance, is one of such cases. A processor, to be preempted, must be executing in the context of a specific process and not in transition between processes. Therefore, the actual process context switching should be an atomic operation as far as process operations are concerned. The kernel context switching code is critical and should be programmed very carefully.

As a closing remark, it should be reminded that there are still sources of additional and unnecessary context switching latency time due to the exclusive use of resources by lower priority processes which are required for higher priority processes to execute. Certainly, these cases deserve a special attention of real-time operating systems designers and should be avoided as much as possible. However, real-time application programmers should also be aware of these problems when distributing the exclusive resource utilization among their processes in order to avoid the generation of an infinite latency time, i.e., a deadlock.

5 - Measurement Methodology

As discussed in the previous section, in an operating system, when the response to an external event is to be executed at the process level, there are three main sources that may cause latency: DMA operations, interrupt operations and process operations. Since latency time of DMA operations cannot be directly measured by software procedures, requiring additional hardware tools that are avoided in this paper, the methodology described in this section is dedicated to the measurement of interrupt acknowledge and context switching latency times.

The latency of an interrupt acknowledge operation is defined as the time it takes from the occurence of the external event which generates the interrupt until the execution of the first instruction of the corresponding interrupt routine. If the first instruction of the interrupt routine is used to read the real-time clock, compounding this information with

the number of seconds and the number of ticks of a second (10 milliseconds, in our case) which have elapsed since a pre-defined epoch, we can obtain a time-stamp. However, we still lack the time-stamp of the external event occurence and, supposedly, require an external hardware to obtain it. But, if the external event can be precisely the one which generates the real-time clock interrupt itself then, by knowing the hardware conditions which generates it, the time-stamp of this external event can be obtained.

From Section 1, we know that the MC68230 generates an interrupt when its internal counter reaches zero and is automatically reloaded with a previously programmed value to generate the next interrupt. Therefore, if the clock interrupt can be used, the time-stamp of the external event is always known and the latency of the interrupt operation can be measured. The remaining condition that should be examined before using the time-stamp of this external event as our starting point is whether the level of the interrupt it generates is appropriate for our measurements.

Clock interrupt is, in general, assigned to the highest maskable priority (in our case, level 6). This means that clock interrupt processing has precedence over all other operations in progress, except DMA operations (actually level 7 is the highest interrupt priority but it is not maskable and, in our case, it is used just for debugging purposes). This fits well in our methodology since the measurements will provide a kind of lower limit for the latency of interrupt operations. Therefore, no other type of interrupt can consistently have a lower latency time than the chosen (clock) interrupt. Now, we just need to have a process that must be started by this interrupt.

Every operating system has a scheduler that will choose, according to a pre-defined procedure, a process for execution as soon as the condition it has been waiting for ("sleeping") does not exist anymore. This condition ceases to exist as a direct or indirect consequence of the occurence of an external event. Therefore, for our measurement purposes, we should tailor a process that just sleeps on this clock event and is directly awakened by its occurence. Upon awakening, it should obtain a time-stamp which is to be compared with the one known for occurrence of the external event. This can be a user process, but the time-stamp will be delayed by the return to user mode and the overhead of another system-call to obtain the time-stamp from the real-time clock. It would be easier to have a kernel process ("daemon") which has direct access to the real-time clock. Fortunately, this process already exists.

Every UNIX-like operating system has a kernel process that is awakened periodically (in general, once a second) to perform some housekeeping chores. In our case it is the process zero, the *swapper/pager*. The use of this process fits well in our methodology for many reasons: it is simple to use since it is not necessary to create an additional process; it has direct access to the hardware (kernel process); it does not have the overhead of returning to user mode; it is called just once a second which causes an extremely low overhead in the execution of the system, provides a steady pace for sampling measurements, and avoids the quick overflow of the measurement area. However, it still has a problem.

Every UNIX-like operating system may execute many "daemons" triggered by the clock interrupt at the turning of a second. The choice of process zero, which, for our measurements to be done, has to be triggered by the clock interrupt as well, may cause latency to be evaluated with some bias because there will be no independence between the sampling period and the activity to be measured. Fortunately, this can be avoided

with a very few additional lines of code.

In order to de-synchronize process zero execution from other processes executing at the turn of a second, we made periods of sampling slightly smaller than one second (in our case, 990 milliseconds, i.e., one "tick" less than a second). With this scheme, the external event time-stamp for generating the clock interrupt, which is going to cause the start of process zero, is obtained in every second at different instants of time inside the period of a second. The increase of approximately 1% in the frequency of process zero execution proved to be negligible as far as system performance is concerned.

Since DMA operations might cause distortions in our measurements (which actually happened), when measurements were being obtained, we kept track of DMA activity during any period of time between (and including) the start of interrupt operation and the start of the process zero execution. For each sampled measurement, it has been registred if any DMA channels, excluding the one used for dynamic memory refresh, which was always active (see Section 2), had pending operations during this period. Measurement sessions were also performed with DMA operating in singleblock and in multiblock (with no limiting factor) to observe the interference of the DMA operation in the measured values.

Another source of latency which was monitored was the current interrupt level at the time of the occurence of the clock interrupt. It is well understood that the start of the execution of a new process can be much delayed if the processing unit is overloaded with I/O and the operating system runs most of the time inside the interrupt routines.

6 - Results of the Measurement Sessions

In order to present an example of the methodology described in the previous section, the system, also described in previous sections, was submitted to different workloads. Under these workloads, the latency time of interrupt acknowledge and process context switching were measured and they are presented in figures and tables.

Two types of workload were used: CPU load and I/O load. CPU load consisted of sequences of compilations of a 200-line program coded in C. One compilation, called through the *cc* command, consisted of the execution of the preprocessor, compiler, assembler, optimizer and loader. For each sequence, a shell script was defined and, during the measurement sessions, they were called to be executed in background.

Four levels of CPU loads were used. Load 0 was used as a reference and corresponds to the idle load, i.e., there were 12 processes loaded in memory but most of them "sleeping" all the time. From those, just the process zero, the *update* (once a second) and *usetime* (once a minute) were regularly awaken. Load 1 consisted of one continuous sequence of compilations while Load 2 and 4 consisted of two and four simultaneously executed sequence of compilations, respectively. It is important to remind that during CPU load measurements all longer programs (compiler, assembler, loader and shell), which are reentrant anyway, were made resident in memory and all swapping/paging activities were disabled.

I/O load consisted of sequences of continuous serial outputs. A serial output was obtained by using the command *cat* on a 200-line source program coded in C and redirecting the standard output of the command to one of the serial lines. Similarly to CPU load, for each sequence, a shell script was defined and, during the measurement

sessions, they were called to be executed in background.

Four levels of I/O loads were also used. Load 0 was used as a reference and corresponds to the idle load as mentioned above. Load 1 consisted of one continuous sequence of serial outputs while Load 2 and 4 consisted of two and four simultaneously executed sequence of serial outputs, respectively.

To account for the influence of kernel preemption on the data obtained, two types of measurement sessions were run. With the preemption on, process context switching could be initiated at any time independently of the fact whether the current process was executing in supervisor mode or not. With the preemption off, process context switching could be initiated only if the current process was executing in user mode or at specific points in supervisor mode.

To account for the influence of DMA operation, two types of measurement sessions were also run. The first one, called singleblock, was run with the operating system reprogramming the DMA and the device (disk) controller every time a single block of data should be transferred. In the second one (multiblock) it was allowed the transfer of multiple blocks from disk in one single DMA operation.

Still to account for the DMA influence, data were also presented excluding the time values measured during a period when there were DMA operations in execution or pending. In no measurement session, however, excluded data made up more than 7% of total data gathered.

Each measurement session lasted about 17 minutes with a total of 1024 values obtained for each latency time (approximately one per second). Many sessions of the same type were run with consistently similar results making us suppose that longer session would render basically the same results. Nonetheless, longer sessions should be used if one wants to increase the certainty that actually no elusive outlier values exist.

Presenting data for these three combinations (preemption, DMA and multiblock), we ended up with eight sets of data for each type of workload. For I/O loads, however, only two sets of data are presented since DMA operations were negligible during such measurement sessions.

Latency time values can be reported by their average and standard deviation, or, like in some articles, by maximum and minimum values. Sometimes, a confidence interval is mentioned with the disclaimer that there were a few outliers. Tables 1 to 10 show the minimum, the median and the maximum values together with average and standard deviation obtained for latency times of each workload during our measurement sessions. However, they do not convey enough information, as far as our interests in latency times are concerned. We argue that a better form of latency time data presentation is required. Preferably a graphical one. It is important to provide an easier form of comparison among situations when the system is submitted to different workloads and, also, permit comparison between different systems under similar workloads.

The highly skewed nature of latency time distributions made us devise a graphic presentation where in the vertical axis the logarithm of the time measured is plotted and in the horizontal axis, the exponential of the percentile. As a reminder, a 80% percentile of 2 millisecons means that 80% of the measured data were bellow 2 milliseconds. The value in the 50% percentile is, by definition, the median.

In these figures, there are two characteristics we should observe carefully. Since the vertical axis, where we plot latency time, is logarithmic (endless, never reaching zero) and shorter latency times mean better systems, the curves corresponding to good real-time systems should be the ones closer to the lower part of the figure.

The other important characteristic is predictability. Since in the horizontal axis we plot percentiles, the curves corresponding to good real-time systems should be the ones which are flatter. A totally flat curve (an horizontal line) means a deterministic latency time, i.e., a measured value which variance is zero. Since our main concern is with the (supposedly) few latency time values which are much higher than the average, this flatness (or lack of it) is much better observed if we plot the percentiles in an exponential scale. In this case, the values on the horizontal axis are continuously amplified as we get to higher values of percentiles. Verify that the median value is plotted close to the left vertical axis of the figures.

In all figures, Load 0 is the idle load. This clearly shows a lower bound for context switching and interrupt acknowledge times. We can see that both curves are reasonably flat. It is interesting to notice that the context switching time for any other non-idle load is, in many cases, approximately twice the value of the idle load context switching time. This is easily understood, since no context saving of the current process is required when an external event, which triggers a new context to be loaded, occurs and the operating system is idling.

These figures should be examined in pairs in order to better understand the influence of each factor in the latency time when all other ones are kept unchanged. Therefore, for instance, comparing Figure 1 with Figure 2, we can notice that the influence of DMA operation when using preemption and singleblock transfers is almost negligible. Anyway, in order to find the sources of latency, independent instrumentation procedures traced the system operation when the higher values of interrupt acknowledge times not caused by DMA operations were obtained and they confirmed, as already predicted, that they were caused by kernel table updates using mutual exclusion primitives.

Comparing Figure 1 with Figure 3 we can see that, when DMA is allowed to work with multiblock transfers, a severe impact can be felt in interrupt acknowledge latency times. This is due to DMA exclusive bus utilization during long periods of time. This can be further confirmed by examining Figure 4 since, excluding measurements taken when DMA was in operation, Figure 4 becomes almost identical to Figure 2. Still comparing Figures 1 and 3, we can see that multiblock DMA operation also causes an increase in the values of context switching times but its impact is much less severe than the one experienced by interrupt acknowledge latency times.

Comparing Figure 1 with Figure 5 we can immediately see why it is almost impossible to use non-preemptible kernels in real-time applications. Even with a light load (Load 1), there might be a high percentage of context switching times which values are totally unacceptable in a real-time environment. Notice, however, that interrupt acknowledge latency times are reasonably unaffected by preemption.

Comparing Figures 5 and 6 we confirm the conclusions taken from comparing Figures 1 and 2: DMA operations have little effect on latency times when DMA operates in singleblock. The comparison of Figures 7 and 8 confirms the conclusions taken by the comparison of Figures 3 and 4: DMA multiblock operations have a much stronger impact

on interrupt acknowledge latency times than on context switching latency times.

Comparing Figures 9 and 10, we can feel the influence of preemption when I/O load is applied. It can be seen that with preemption and with just one I/O load the system is capable of operating with still acceptable context switching time values. With more than two I/O loads, though preemption helps, the system becomes clearly overloaded. Interrupt acknowledge latency times are clearly unafected by preemption or any type of I/O load because the latency time we are measuring corresponds to the acknowledgement of the highest priority level interrupt (clock) and this level of priority is not used by any I/O load.

7 - Conclusions

Real-time applications are becoming increasingly common. They require powerful hardware and real-time operating systems to perform appropriately. Despite their time-sharing origin, UNIX-like operating systems are probably to become at least one of the main choices to support such real-time applications. Among the characteristics a real-time operating system must have, a short latency time is one of the most important ones.

In this paper we presented a methodology which can easily be applied to any UNIX-like operating system in order to evaluate its interrupt ackowledge and context switching latency times. This methodology can also be combined with some instrumentation and can be used, as we have actually been doing in the development of TROPIX, to evaluate and improve the responsiveness characteristics of operating systems. The light-load nature of the methodology allows its implementation in normal use operating system without users being aware of its existence. Valuable data can be collected when, in this case, the operating system is submitted to the actual installation workload.

From our experience in the performance evaluation of the TROPIX operating system, as show in this paper, we feel that real-time applications (and their required real-time operating systems) may strongly benefit from computer architectures with multiple processors which can quickly communicate with each other (through a common global memory, for instance) and can quickly load and unload their internal states to allow shorter context switching latency times. Unfortunately, this is not the case of many current RISC processors. However, with the announcement of new technologies which can permit a faster memory access by a much higher number of processors (RamLink, for instance) than current multiprocessing systems do and with processors becoming less and less expensive, we might have in the near future the possibility of building systems in which each processor may control a single operation (process or thread, for instance) with very little or not at all additional interrupt or DMA interference. In these cases, the latency time would be reduced to hardware latency time only. And better, some of the real-time operating systems that have already been presently designed with multiprocessing capabilities in mind could immediately be used to control these new architectures.

Acknowledgements

I would like to thank Dr. Jerome Feldman and Dr. Domenico Ferrari for receiving me as a visiting scholar in the Tenet group of the International Computer Science Institute (ICSI), which is affiliated to the University of California at Berkeley, U.S.A. I thank the Brazilian Government Research Funding Agency - CNPq, the Instituto de Matemática and the Núcleo de Computação Eletrônica of the Universidade Federal do Rio de Janeiro, Brazil, for funding my stay at ICSI. I thank Unitec Informática for providing the EBC-32010 computer and the TROPIX operating system. I thank Bryan Costales and Alex Nguyen of ICSI staff for helping me with the paperwork and the electrical work, respectively, for installing the EBC-32010 in ICSI. Finally, I have to acknowledge the many hours of profitable discussion which I had with my coleague Dr. Pedro Salenbauch while we were developing the TROPIX operating system.

References

[Bach86] Bach, M. *The Design of the UNIX Operating System*, Prentice-Hall, New Jersey, 1986.

[EBCto87] *EBC-32010: Teoria de Operação,* EBC, 1987.

[EBCde87] *EBC-32010: Diagramas Elétricos*, EBC, 1987.

[FaSa90] Faller, N., and Salenbauch, P., "TROPIX: Um Sistema Operacional Unix-like para Tempo-Real", *Anais do XXIII Congresso Nacional de Informática*, Sucesu90, Rio de Janeiro, August 1990.

[FaSa89] Faller, N., and Salenbauch, P., "PLURIX: A Multiprocessing UNIX-like Operating System", *Proceedings of the Second IEEE Workshop on Workstation Operating Systems,* IEEE Computer Society Press, Washington, D.C., September 1989.

[Faal84] Faller, N. et alii, "O Projeto PEGASUS-32X/PLURIX", *Anais do XVII Congresso Nacional de Informática,* Sucesu84, Rio de Janeiro, November 1984.

[Fual91] Fuhrt, B., Grostick, D., Gluck, D., Rabbat, G, Parker, J., and McRoberts, M., "Real-Time Unix Systems - Design and Application Guide", Kluwer Academic Publishers, 1991.

[KaSZ92] "Realtime Scheduling in SunOS 5.0", Khanna, S., Sebré, M., and Zolnowski, J., *Proceedings of the Winter 1992 USENIX Conference*, San Francisco, January 1992.

[MOTfr90] *M68000 Family Reference Manual*, Motorola, 1990.

[MOTpr84] *M68000 16/32-bit Microprocessor: Programmer's Reference Manual*, Motorola, 1984.

[StRa88] Stankovic, J. A. and Ramamrithan, K., *Tutorial on Hard Real-Time Systems, IEEE Computer Society,* 1988.

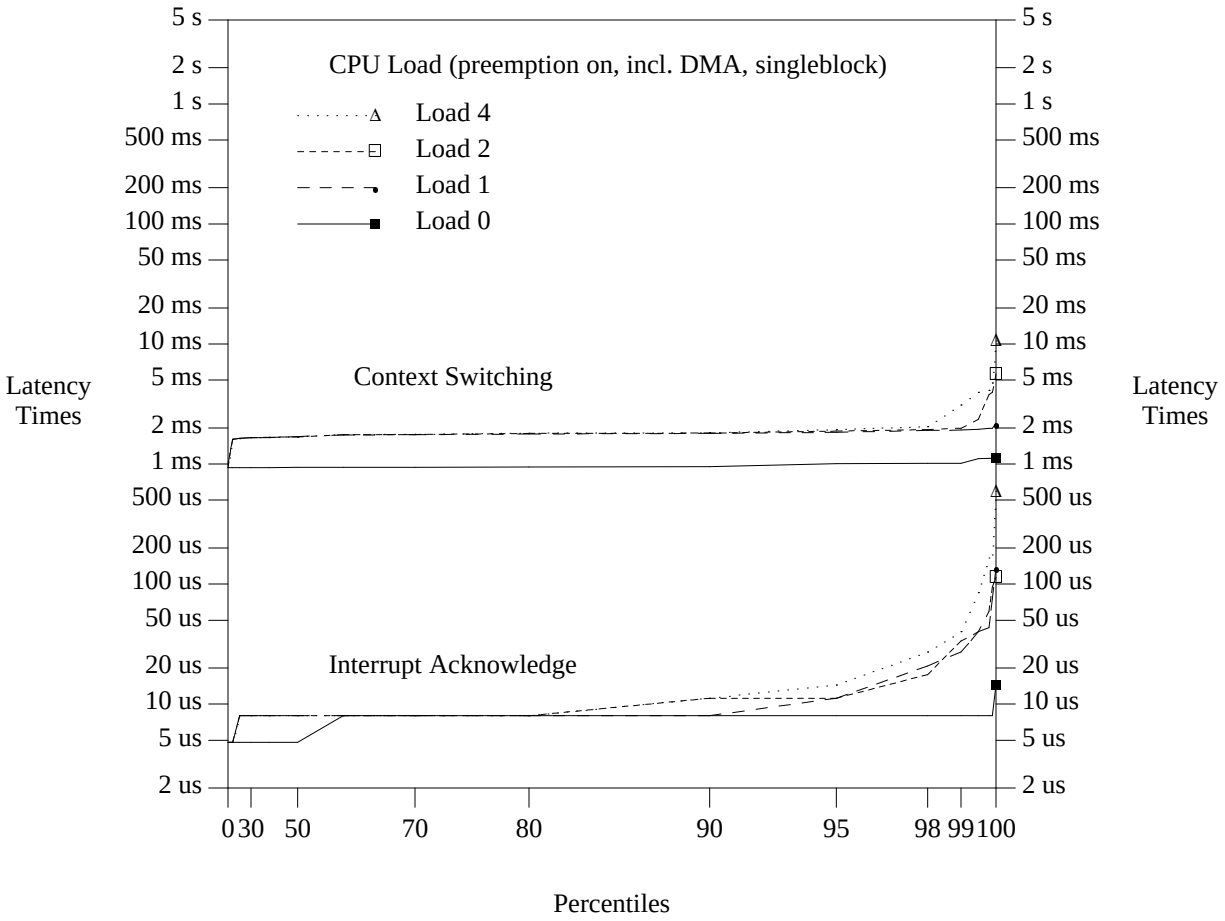

Figure 1

CPU Load (preemption on, incl. DMA, singleblock)					
Interrupt Acknowledge (microseconds)					
	min.	med.	max.	average	std. dev.
load 0	4.8	4.8	14.4	9.89	1.62
load 1	4.8	8.0	136.0	12.13	6.48
load 2	4.8	8.0	113.6	12.23	6.47
load 4	4.8	8.0	593.6	13.59	23.50
Context Switching (microseconds)					
	min.	med.	max.	average	std. dev.
load 0	933	936	1122	943	27
load 1	994	1694	2184	1704	87
load 2	939	1682	5614	1716	235
load 4	936	1682	10610	1740	431

Table 1

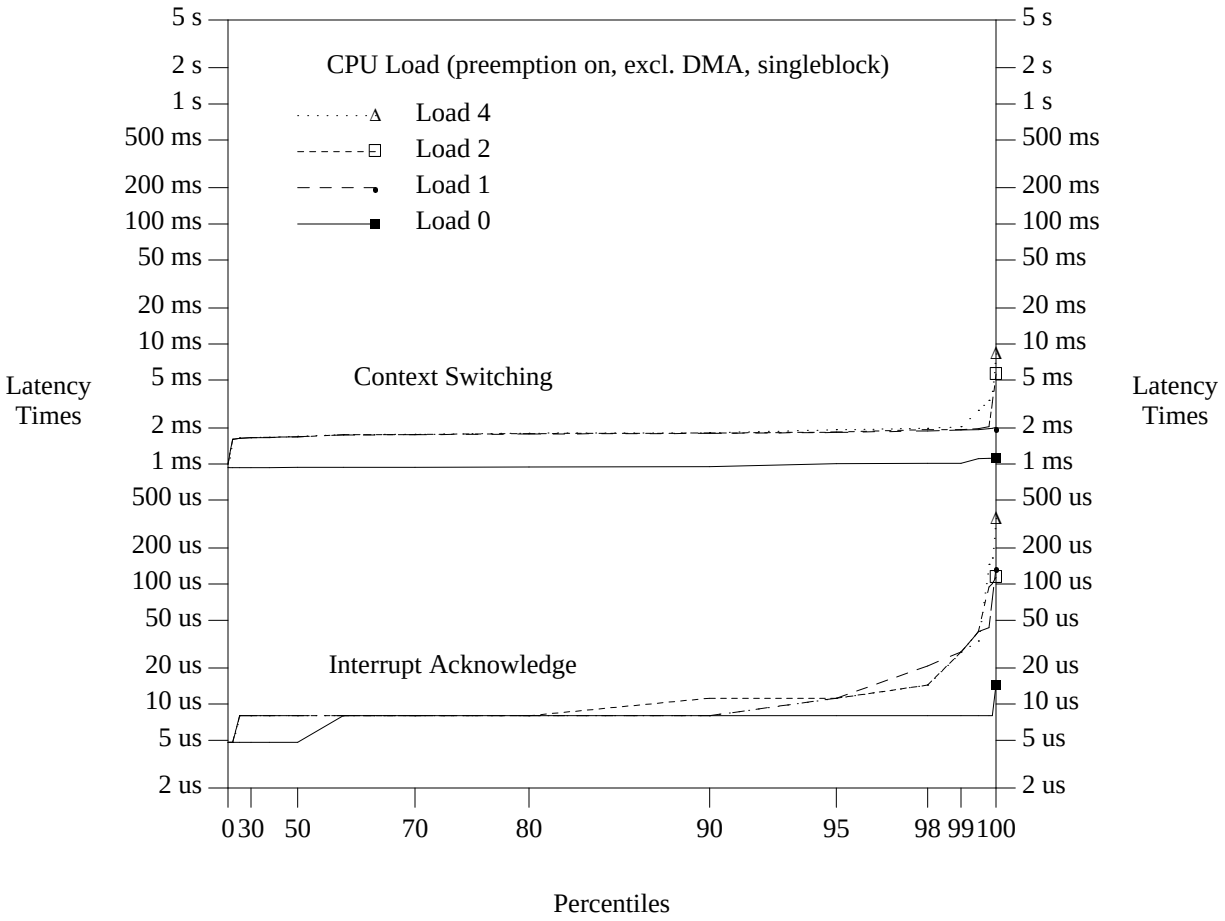

Figure 2

CPU Load (preemption on, excl. DMA, singleblock)					
Interrupt Acknowledge (microseconds)					
	min.	med.	max.	average	std. dev.
load 0	4.8	4.8	14.4	9.89	1.62
load 1	4.8	8.0	136.0	12.10	6.41
load 2	4.8	8.0	113.6	12.13	6.13
load 4	4.8	8.0	350.4	12.59	13.84
Context Switching (microseconds)					
	min.	med.	max.	average	std. dev.
load 0	933	936	1122	943	27
load 1	994	1691	2018	1704	86
load 2	994	1682	5614	1710	172
load 4	936	1698	8267	1725	274

Table 2

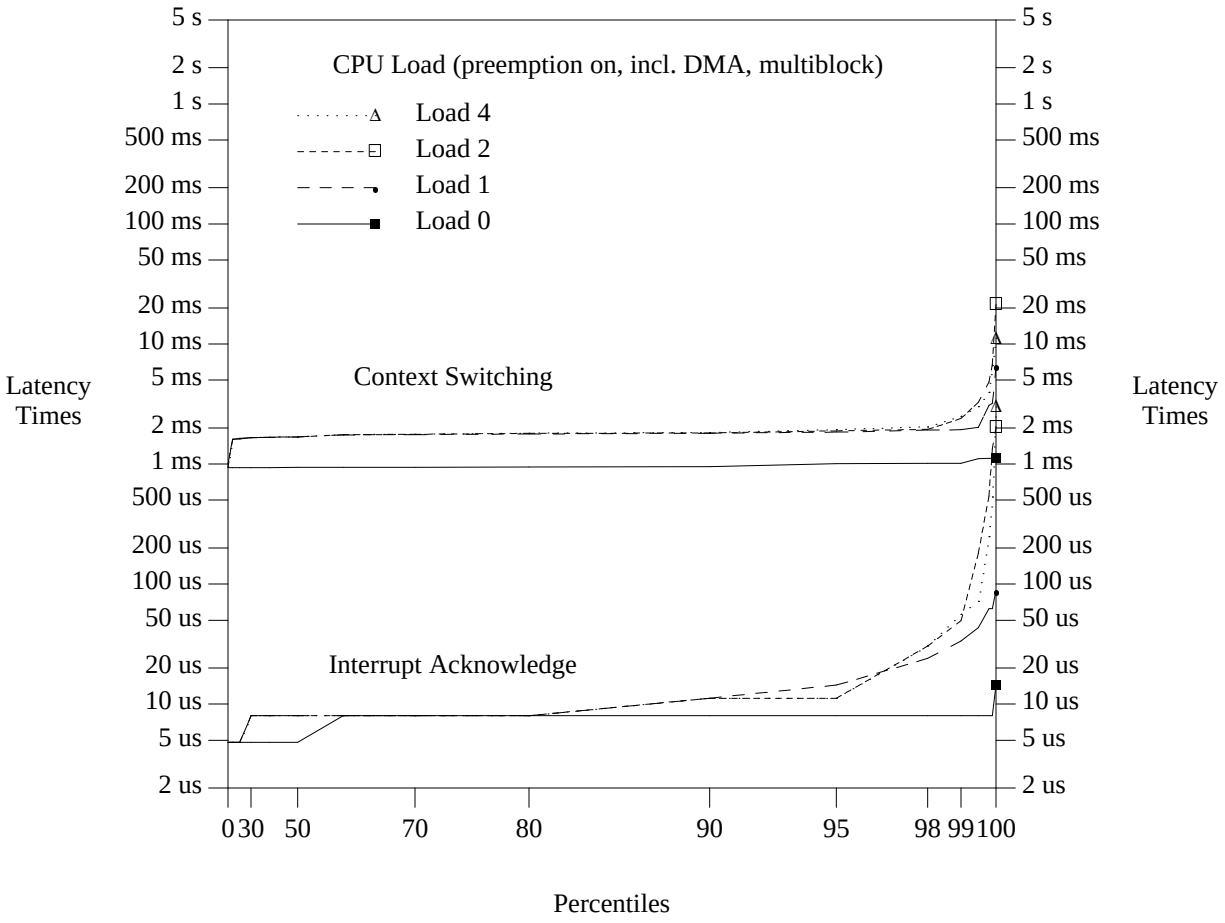

Figure 3

CPU Load (preemption on, incl. DMA, multiblock)					
Interrupt Acknowledge (microseconds)					
	min.	med.	max.	average	std. dev.
load 0	4.8	4.8	14.4	9.89	1.62
load 1	4.8	8.0	88.0	11.94	5.83
load 2	4.8	8.0	2014.4	17.74	90.64
load 4	4.8	8.0	2993.6	16.85	99.96
Context Switching (microseconds)					
	min.	med.	max.	average	std. dev.
load 0	933	936	1122	943	27
load 1	994	1682	6546	1712	197
load 2	939	1678	21413	1739	712
load 4	942	1682	11176	1737	478

Table 3

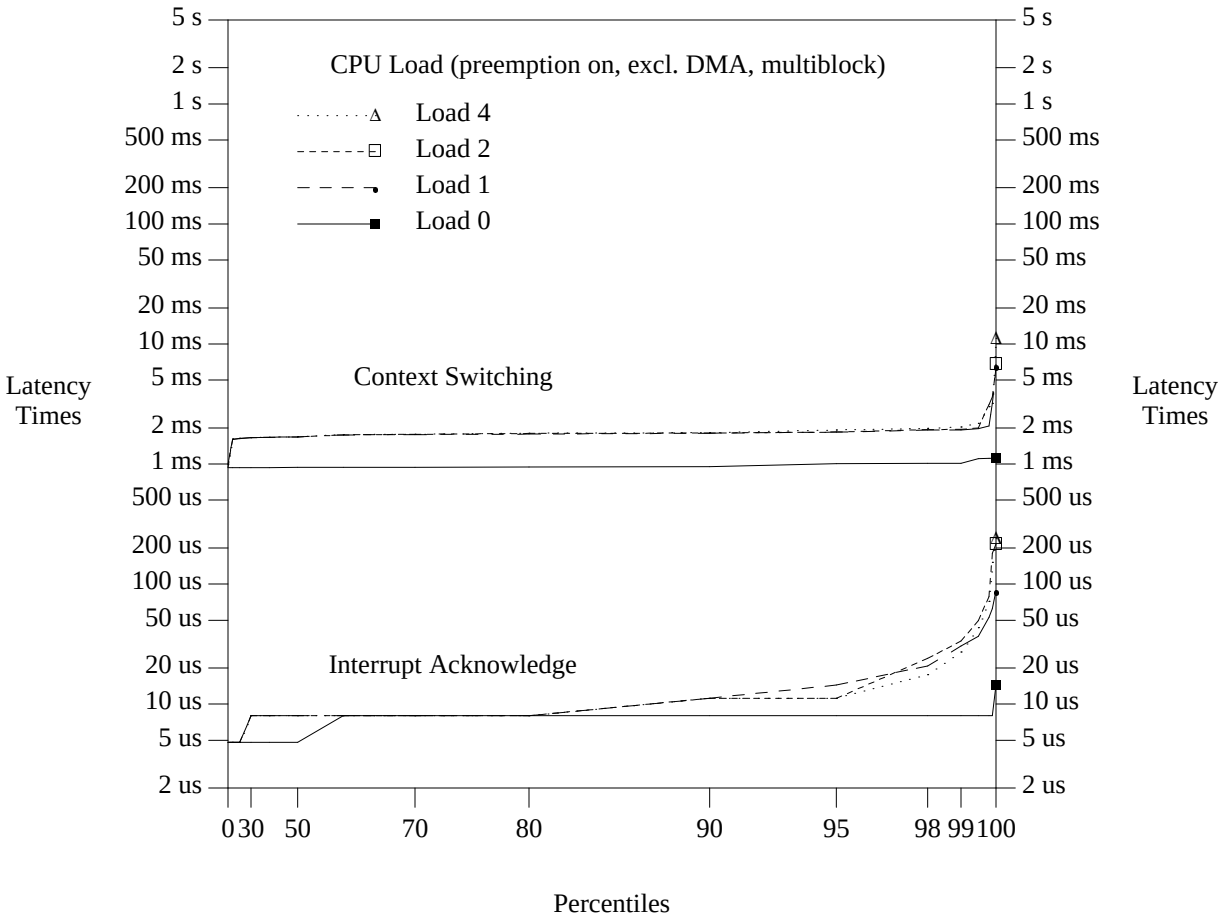

Figure 4

CPU Load (preemption on, excl. DMA, multiblock)					
Interrupt Acknowledge (microseconds)					
	min.	med.	max.	average	std. dev.
load 0	4.8	4.8	14.4	9.89	1.62
load 1	4.8	8.0	88.0	11.89	5.59
load 2	4.8	8.0	216.0	12.18	10.13
load 4	4.8	8.0	238.4	12.00	9.87
Context Switching (microseconds)					
	min.	med.	max.	average	std. dev.
load 0	933	936	1122	943	27
load 1	994	1682	6546	1710	191
load 2	939	1678	6754	1702	228
load 4	994	1685	11176	1721	344

Table 4

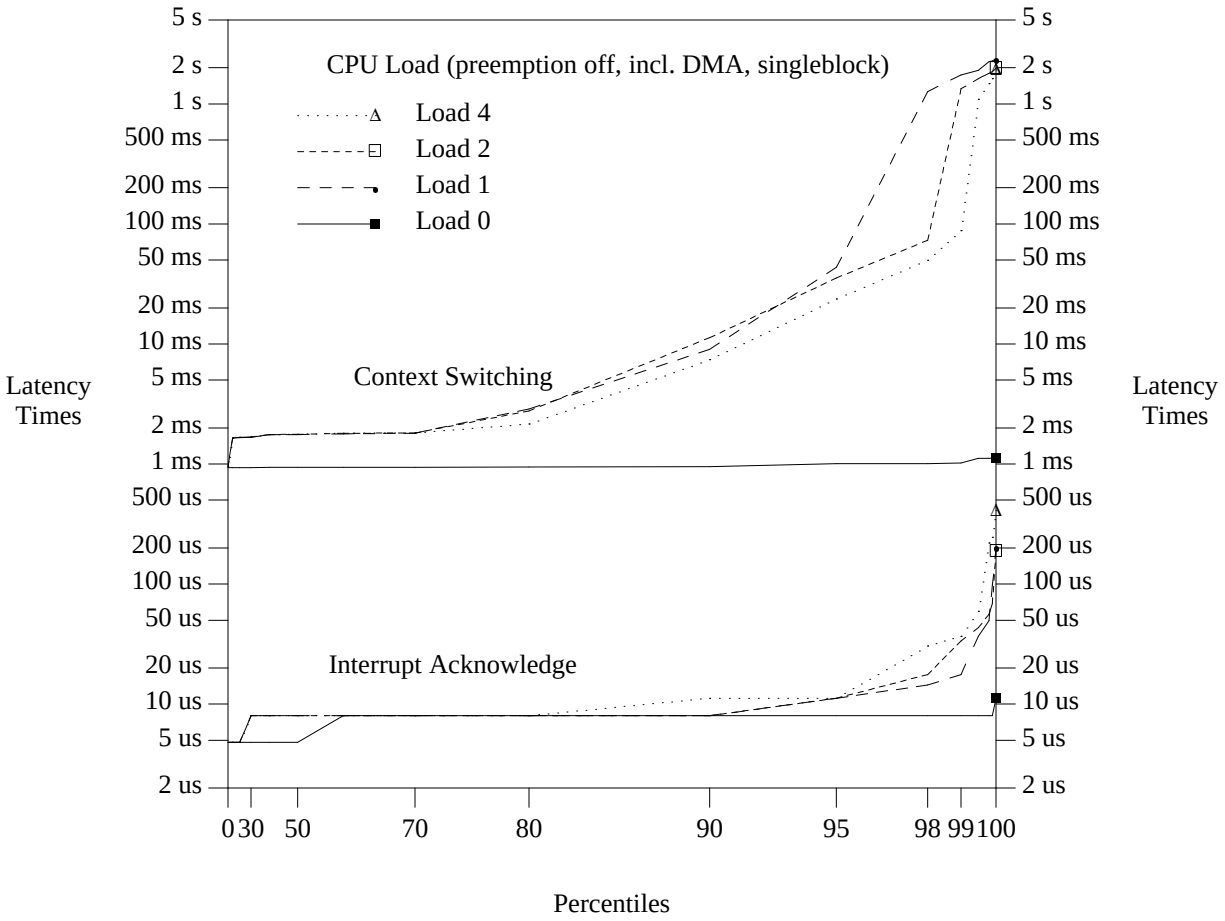

Figure 5

CPU Load (preemption off, incl. DMA, singleblock)					
Interrupt Acknowledge (microseconds)					
	min.	med.	max.	average	std. dev.
load 0	4.8	4.8	11.2	9.92	1.60
load 1	4.8	8.0	203.2	11.70	8.44
load 2	4.8	8.0	187.2	11.68	7.44
load 4	4.8	8.0	408.0	12.93	20.71
Context Switching (microseconds)					
	min.	med.	max.	average	std. dev.
load 0	933	936	1122	943	27
load 1	942	1758	2376561	53826	281996
load 2	946	1768	1974965	29499	191326
load 4	942	1768	1915694	17015	135660

Table 5

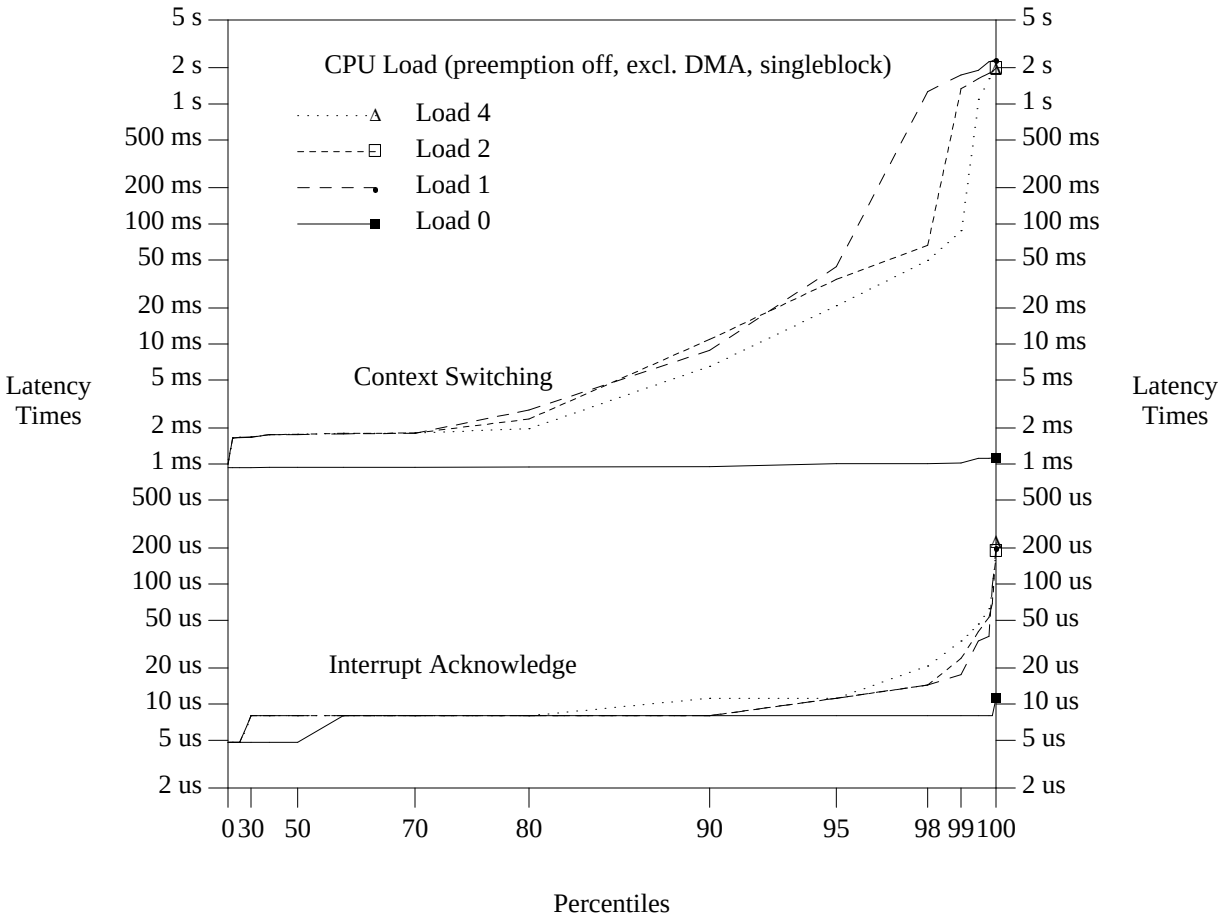

Figure 6

CPU Load (preemption off, excl. DMA, singleblock)					
Interrupt Acknowledge (microseconds)					
	min.	med.	max.	average	std. dev.
load 0	4.8	4.8	11.2	9.92	1.60
load 1	4.8	8.0	203.2	11.68	8.41
load 2	4.8	8.0	187.2	11.59	7.22
load 4	4.8	8.0	219.2	11.93	8.70
Context Switching (microseconds)					
	min.	med.	max.	average	std. dev.
load 0	933	936	1122	943	27
load 1	994	1758	2376561	54810	284705
load 2	994	1768	1974965	28177	188595
load 4	1090	1768	1915694	16058	131431

Table 6

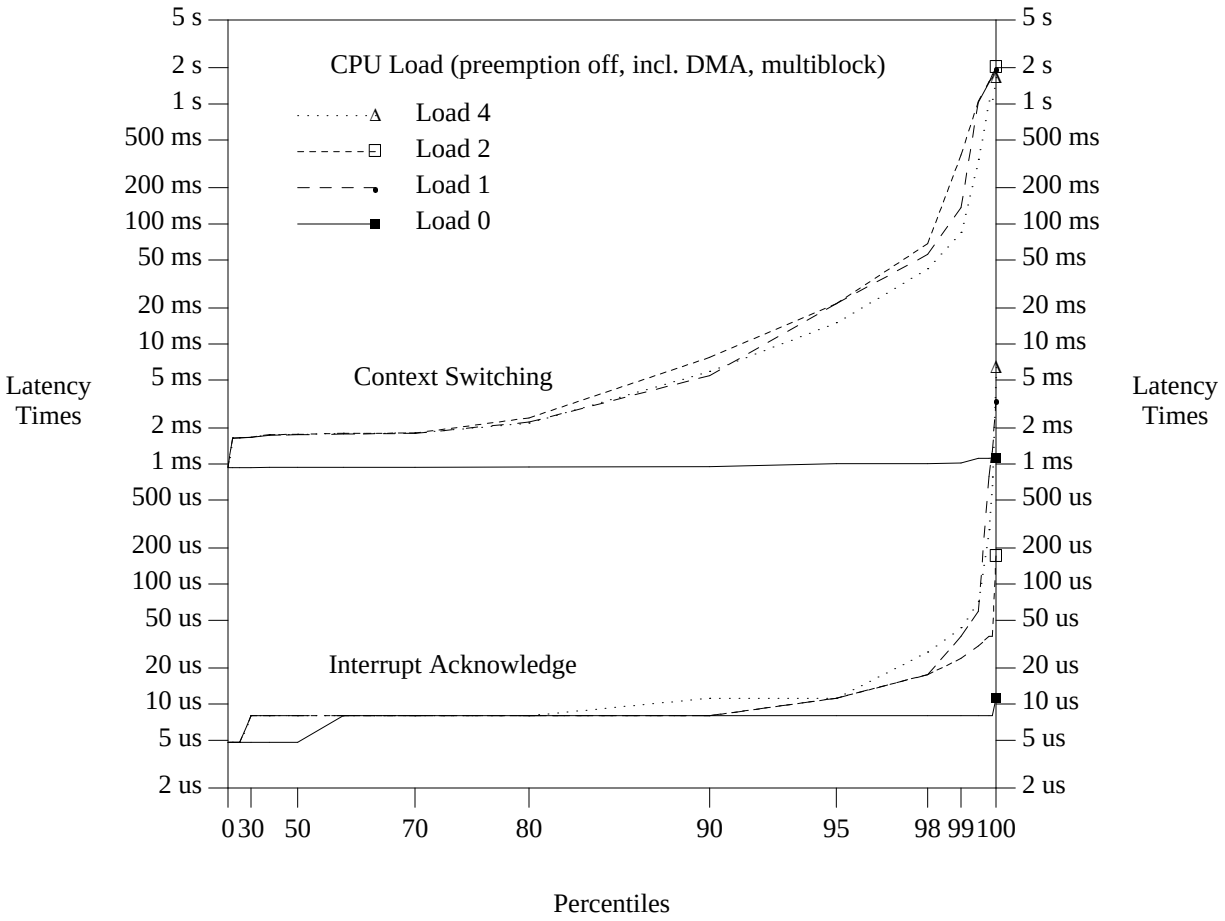

Figure 7

CPU Load (preemption off, incl. DMA, multiblock)					
Interrupt Acknowledge (microseconds)					
	min.	med.	max.	average	std. dev.
load 0	4.8	4.8	11.2	9.92	1.60
load 1	4.8	8.0	3460.8	18.61	125.84
load 2	4.8	8.0	171.2	11.63	6.31
load 4	4.8	8.0	6340.8	22.53	223.85
Context Switching (microseconds)					
	min.	med.	max.	average	std. dev.
load 0	933	936	1122	943	27
load 1	940	1755	1993531	15588	127405
load 2	946	1765	1997550	18056	134663
load 4	940	1765	1657474	11072	90162

Table 7

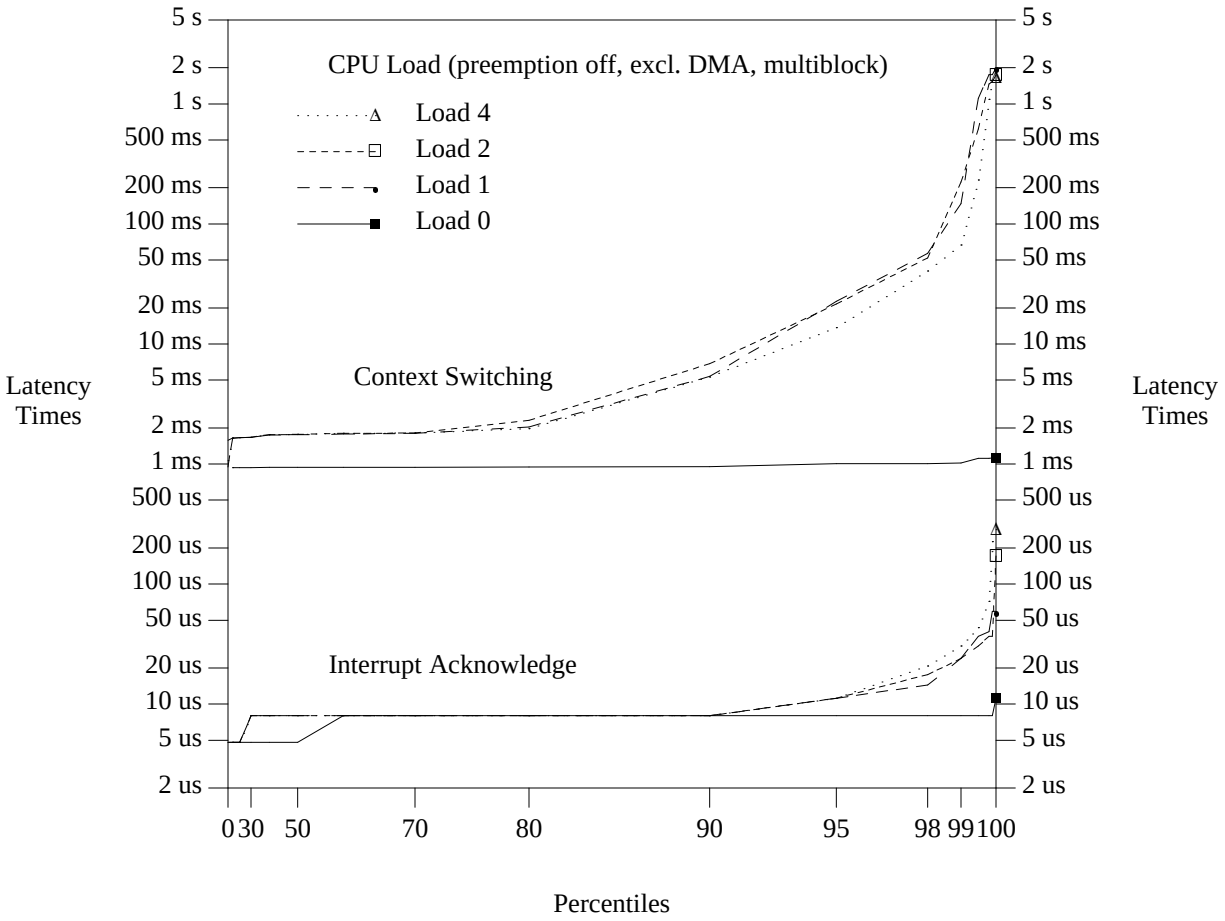

Figure 8

CPU Load (preemption off, excl. DMA, multiblock)					
Interrupt Acknowledge (microseconds)					
	min.	med.	max.	average	std. dev.
load 0	4.8	4.8	11.2	9.92	1.60
load 1	4.8	8.0	59.2	11.45	3.94
load 2	4.8	8.0	171.2	11.62	6.33
load 4	4.8	8.0	283.2	12.09	12.06
Context Switching (microseconds)					
	min.	med.	max.	average	std. dev.
load 0	933	936	1122	943	27
load 1	942	1755	1993531	15923	129294
load 2	1576	1765	1745483	15630	118748
load 4	1064	1765	1657474	9329	83273

Table 8

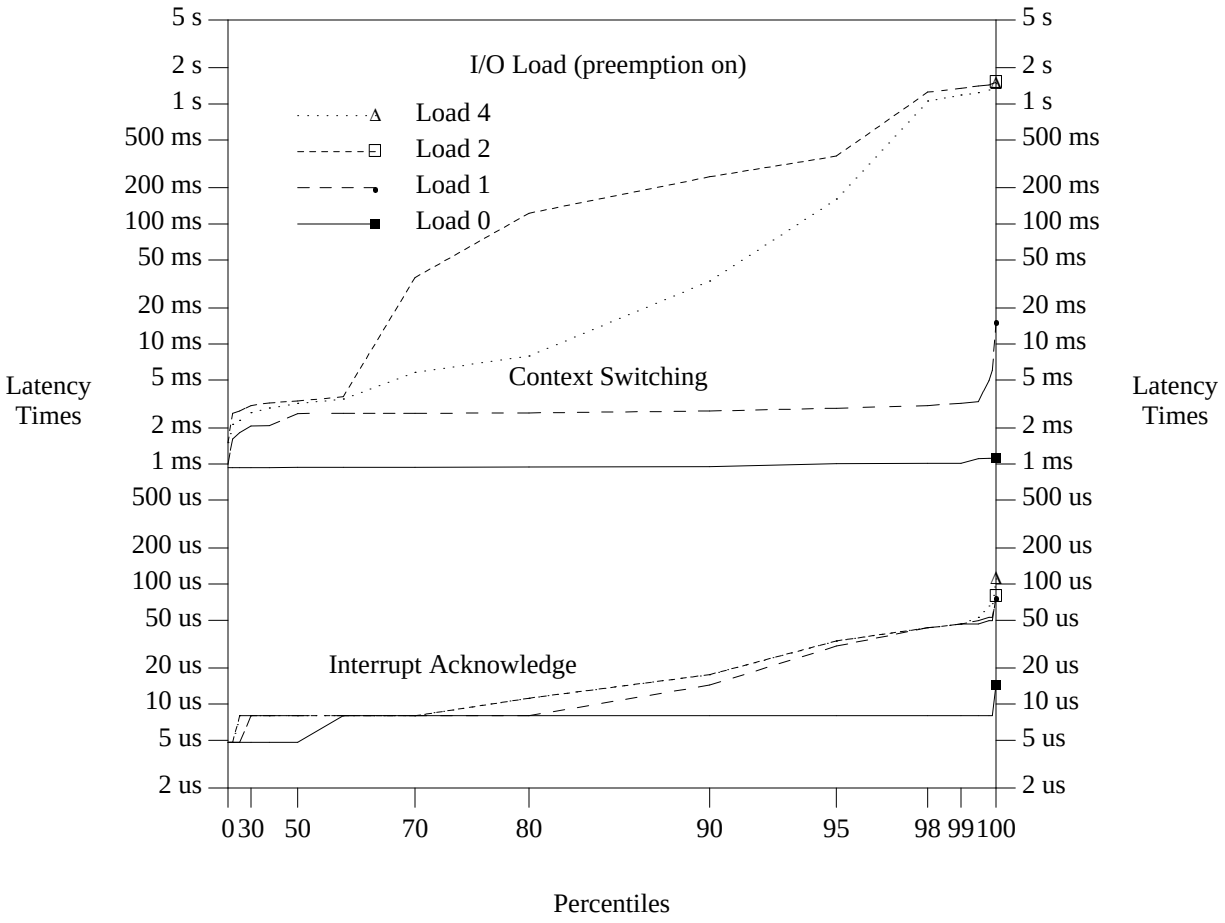

Figure 9

I/O Load (preemption on)					
Interrupt Acknowledge (microseconds)					
	min.	med.	max.	average	std. dev.
load 0	4.8	4.8	14.4	9.89	1.62
load 1	4.8	8.0	78.4	13.40	8.14
load 2	4.8	8.0	78.4	14.38	8.70
load 4	4.8	8.0	110.4	14.55	9.68
Context Switching (microseconds)					
	min.	med.	max.	average	std. dev.
load 0	933	936	1122	943	27
load 1	994	2635	15598	2339	645
load 2	1506	3352	1510565	101503	262710
load 4	1506	3195	1471656	42334	182026

Table 9

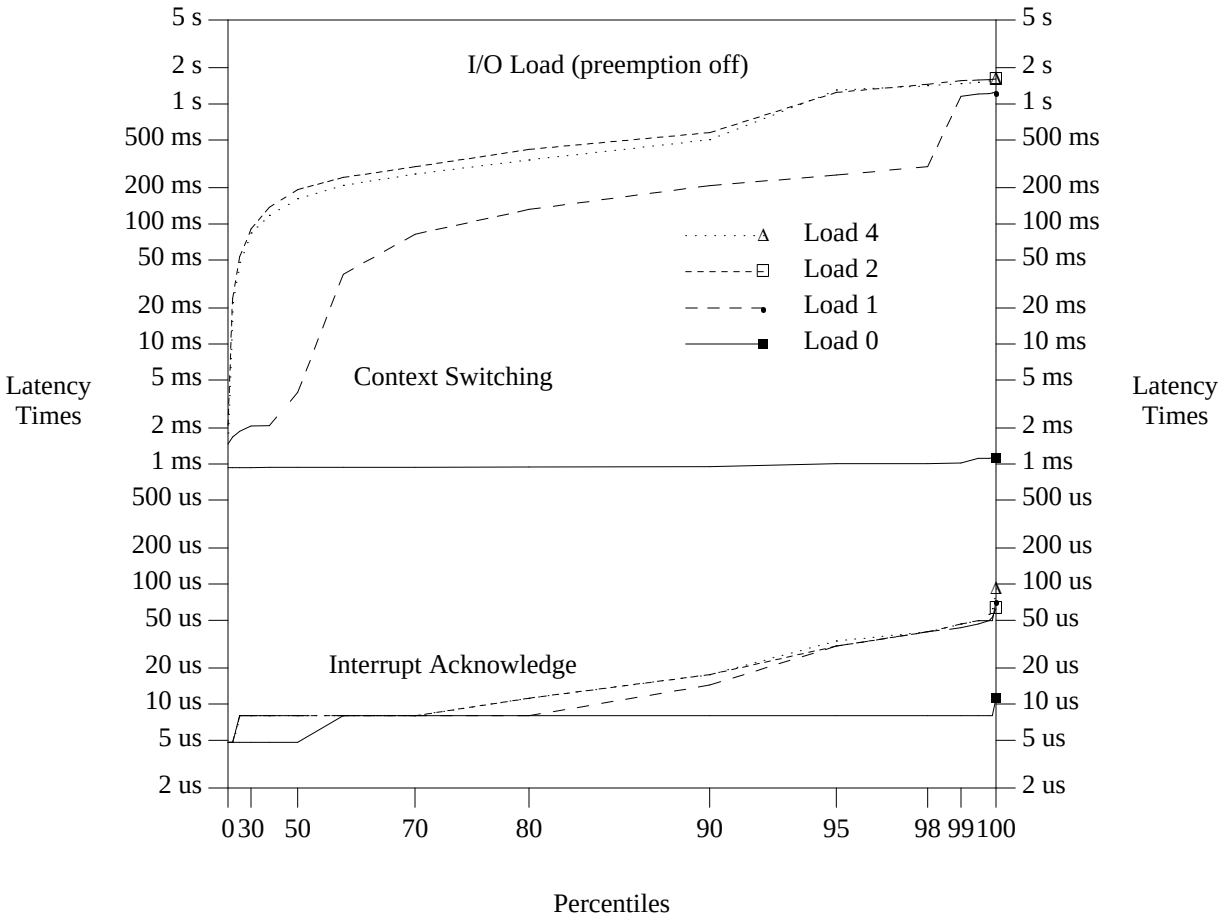

Figure 10

I/O Load (preemption off)					
Interrupt Acknowledge (microseconds)					
	min.	med.	max.	average	std. dev.
load 0	4.8	4.8	11.2	9.92	1.60
load 1	4.8	8.0	72.0	13.54	7.95
load 2	4.8	8.0	62.4	14.24	8.34
load 4	4.8	8.0	91.2	14.34	8.77
Context Switching (microseconds)					
	min.	med.	max.	average	std. dev.
load 0	933	936	1122	943	27
load 1	1470	3947	1268133	79285	173112
load 2	2081	193330	1608939	289994	341812
load 4	1816	161941	1629448	262388	336783

Table 10