International Journal on Recent and Innovation Trends in Computing and Communication ISSN: 2321-8169
Volume: 4 Issue: 8 147 - 155

RTS hypervisor qualification for real-time systems

1Hasan Fayyad-Kazan, 2Luc Perneel, 3Long Peng, 4Fei Guan, 5,6Martin Timmerman
1PostDoc Researcher, 2, 3, 4 PhD Candidates, 5Professor and EmSlab director
Department of Electronics and Informatics, Vrije Universiteit Brussel, Pleinlaan 2- 1050 Brussel, Belgium
E-mails: {1hafayyad, 2luc.perneel, 3longpeng, 4feiguan, 5martin.timmerman}@vub.ac.be
6CEO Dedicated Systems Experts NV/SA Belgium m.timmerman@dedicated-systems.com

Abstract:-Virtualization is a synonym for the server and cloud computing arena. Recently, it started to be also applied to real-time embedded systems with timing constraints. However, virtualization products for data centers and desktop computing cannot be readily applied to embedded systems because of differences in requirements, use cases, and computer architecture.
Bridging the gap between virtualization and real-time requirements imposes the need of real-time virtualization products. Therefore, some embedded software manufacturers have built several real-time hypervisors specialized for embedded systems.
Currently, there are several commercial ones such as Greenhills INTEGRITY MultiVisor, Real-Time Systems (RTS) GmbH Hypervisor, Tenasys eVM for Windows, National Instruments Real-Time Hyper Hypervisor, and some others.
This paper provides the behavior and performance results of evaluating RTS hypervisor and gives advices of its use for soft or hard real-time embedded systems.

Keywords: *Hypervisor, Real-time, Real-Time Systems GmbH, Virtualization*

*****

1. INTRODUCTION

Virtualization is one of the hottest trends in information technology today. It is a mechanism that allows the physical machine resources to be shared among different virtual machines (VMs) via the usage of a software layer called hypervisor or Virtual Machine Monitor (VMM). It is a fundamental component in cloud computing because it provides numerous guest VM transparent services, such as live migration, high availability, rapid checkpoint, etc [1].

Virtualization in the server and desktop world has already matured, with both software and hardware solutions available for several years [8, 9, 10, 11, 12, 13]. In recent years, the introduction of multi-core to embedded systems has brought the availability of increased computing power to embedded systems. Virtualization on embedded systems has only been explored for the past years [14, 15, 16, 17], and is an area of ongoing research which is likely to become more widespread in the next few years.

Unlike the server world, where VMs typically run multiple copies of the same (or similar) operating systems, VMs in the embedded space are more likely heterogeneous, running different classes of operating systems: a real-time operating system (RTOS) for traditional embedded real-time purposes, and a general-purpose (fully-featured) operating system to support complex applications such as user interfaces [3].

Embedded virtualization is already deployed in several domains such as avionics systems and industrial automation where a strong emphasis on real-time performance is required [2]. Also, it is used for soft real-time applications such as media-based ones and even satellite communication systems.

Virtualizing such systems means inserting a new layer between the hardware and Operating System (OS), and thus adding potentially extra overhead. Some of these systems' applications do not demand hard real-time guarantees, but require that the underlying virtualization layer supports both low latency and provide adequate computational resources for completion within a reasonable timeframe [4]. Both these aspects are intimately intertwined with the logic of the hypervisor scheduler [4]. Thus, the performance overhead introduced by the virtualization layer should be limited or minimalistic, and very importantly the system should remain deterministic. Our contribution germinated from this point, and we want to benchmark the performance (latencies that can happen in a VM) of several embedded virtualization solutions

.

In order to accomplish our aim, we did contact several vendors to participate in this benchmark but unfortunately only Real-Time Systems GmbH accepted for now the evaluation of their hypervisor: "Real-Time Systems GmbH Hypervisor". This hypervisor is intended to provide hard real-time support for virtualized RTOSs as published by the vendor. Our results should confirm or refute this.

This paper is organized as follows: Section 2 describes RTS architecture; section 3 explains the experimental setup used for our evaluation; section 4 presents the evaluation test metrics together with their results when applied to a non-virtualized system; section 5 explains the use cases used to test RTS hypervisor together with the results; section 6 provides a comparative summary of the RTS results compared to the non-virtualized system; and finally a conclusion.

2. RTS HYPERVISOR

Real-Time Systems' (RTS) Hypervisor is a software abstraction layer that partitions the hardware resources of a standard x86, multicore-processor execution platform in such a way that multiple operating systems (RTOS and/or General Purpose Operating System) can run concurrently and in complete independence of one another [5].

The number of operating systems that can run simultaneously is limited only by the number of available logical CPUs. As such, the RTS Hypervisor does not partition anything in the time domain on a certain processing resource, but makes the processing resource fully available for the virtualized operating system. Therefore, they run at full speed and full efficiency. They are enough isolated from one another

so that an OS can be booted or rebooted without slowing or compromising the ongoing activities of other operating systems [5].

Figure 1 below describes the RTS Hypervisor architecture.

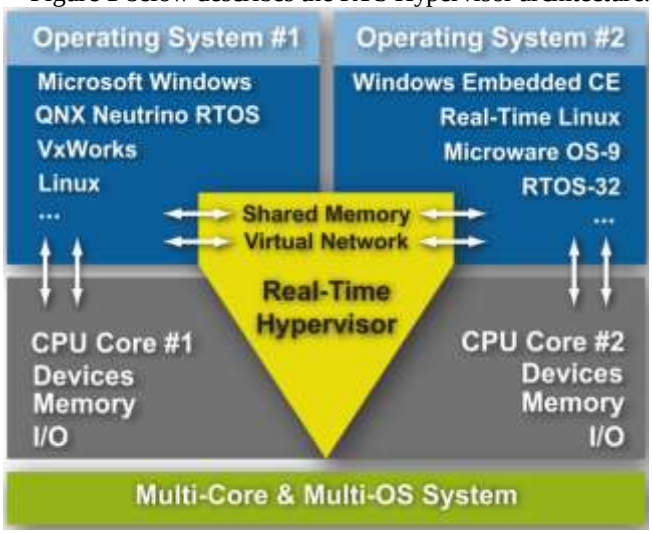

Figure 1: RTS hypervisor architecture [5]

The RTS Hypervisor supports two modes of operating system execution: virtualized and non-virtualized (privileged).

The virtualized mode is intended for General Purpose OSs (e.g., Microsoft Windows) that have no hard real-time requirements. Virtualization is required to guarantee that they cannot in any way influence operating systems that run in other hardware partitions [5].

The privileged (non-virtualized) mode is intended for OSs that offer hard real-time performance (i.e. RTOS). To guarantee deterministic behavior and latency times, such systems require direct access to the hardware. An operating system that runs in privileged mode does not run in a virtual machine; it runs instead in its own hardware partition. This fully protects it and its resources from all other unrelated system events from non-privileged partitions [5]. However the protection is asymmetric: a privileged mode operating system could still impact non-privileged operating systems running on the same system as it runs non-virtualized. This is a tradeoff taken to be able to guarantee real-time latencies.

For the scheduling part of RTS, as mentioned before, it uses partitioning which allows a guest OS to run directly on the physical core and as such the latter is not shared between different guest OSes. As a result, there is no contention within the time domain on processing resources, which makes scheduling extremely simple as there are no scheduling decisions to be taken at all.

3. EXPERIMENTAL SETUP

RTS Hypervisor version 4.1 (latest version at the time of writing this paper) is evaluated here. Linux PREEMPT-RT 3.8.13-rt11 is the OS used in the VM where our testing suite is performed. This VM is called throughout this paper as Under Test Virtual Machine (UTVM). The RT Linux version used is shipped together with the RTS hypervisor from Real-Time Systems GmbH. It has a small difference compared to the classical Vanilla Linux with RT extensions which is an extra network driver to have a virtual network with the GPOS, which in turn is the OS used to control the whole system. There are no fundamental changes in the kernel.

The testing results presented in this pear are applicable only to these mentioned hypervisor and OS versions as other versions may have other significant performance results.

Our testing software uses mlockall() in the Linux kernel to assure that all memory is locked into memory. Further, the application was statically linked and started from a RAM disk (tmpfs) to avoid swapping out read-only code pages.
The hardware platform used for conducting the tests has the following characteristics: Intel® Desktop Board DH77KC, Intel® Xeon® Processor E3-1220 v2 with 4 cores each running at a frequency of 3.1 GHz, and no hyper-threading support. The cache memory size is as follows: each core has 32 KB of L1 data cache, 32KB of L1 instruction cache and 256 KB of L2 cache. L3 cache is 8MB accessible by all cores. The system memory is 8 GB.

4. TESTING PROCEDURES AND RESULTS ON NON-VIRTUALIZED SYSTEM

This evaluation is performed using several tests. These tests are divided into two categories: short-term and long-term tests.

The short-term tests are mainly intended to show the behavior of the system. In these test, a limited number of samples (128000) are captured to simulate the case of embedded systems where a small RAM buffer is available.

The long-term tests are done for hours and intended to provide the probabilistic worst case that could happen in the system. The aim is to verify the determinism and predictability of the system.

In the short-term test, a memory buffer is filled with a number of samples, while the long-term test uses the same measurement system as short-term one but counts the number of samples occurring during a certain interval. In such approach, the measured delay values are counted in binary based bins.

Although the test metrics explained below are mostly used to examine the real-time performance and behavior of RTOSs on bare-machines [6] [7], they are useful to be used in other OS test cases. Moreover, virtualization together with real-time support emerges to be used in an increasing amount of use cases, varying from embedded systems to enterprise computing. Therefore, these tests are a good fit for this paper evaluation.

A. Measuring process,results overhead and precision

In order to do our measurements, a tool or instrument needs to be used. The cheapest solution is to use an on processor chip timer running on the constant frequency of the processor clock giving as a value the number of cycles occurred on the processor. Its value is set to zero every time the processor is reset. This timer is called Time Stamp Counter (TSC). It is a 64-bits register present on all x86 processors and has an excellent high-resolution. Recent Intel processors include a constant rate TSC. This can be verified by checking,

using a Unix-Like OS, the presence of the "constant_tsc" flag in Linux's /proc/cpuinfo. The processor used in our work has this flag. With these processors, the TSC reads at the processor's maximum rate regardless of the actual CPU running rate and thus the timer is not impacted by power saving mechanisms.

In order to access the TSC, the programmer has to call the Read Time-Stamp Counter (RDTSC) instruction from assembly language.

The used tracing system generates an overhead due to reading the TSC values and saving them in RAM buffers. In order to calculate this overhead, an initial test was done in which the tracing overhead was calculated. This overhead is on average 0.0084 µs, and is subtracted from the results of the following tests.

Moreover, each individual measurement have of course its uncertainty coming from different factors such as counter resolution, caching (presence of our measurement instruction in the instruction cache), and frequency stability of the quartz clock of the processor.

After conducting several tests, we can confirm that the results provided in this paper have an uncertainty of 0.05 µs. The details of how these values were obtained can be found in [18].

B. Testing metrics

Below is an explanation of the evaluation tests. Note that the tests are initially done on a ***non-virtualized machine*** (further called Bare-Machine) as a reference, using the same OS as the UTVM.

1) Clock tick processing duration

Like any time-sharing system, Linux allows the simultaneous execution of multiple threads by switching from one threads to another in a very short time frame. The Linux scheduler supports multiple scheduling classes, each using different scheduling algorithms. For instance, there are the two real-time (strict priority) scheduling classes SCHED_FIFO and SCHED_RR, a normal scheduling class (SCHED_OTHER) using dynamic and thus non strict priorities, and finally the SCHED_BATCH class for background threads. The prioritization between these scheduling classes is strict as well, where the SCHED_FIFO/SCHED_RR are using the same highest priority, followed by the SCHED_OTHER and finally, at the lowest priority by the SCHED_BATCH class.

As in the tests we perform only measurements using threads of different priorities, we use the SCHED_FIFO scheduling class in all these tests.

To be able to use timeouts, sleeps, round robin scheduling, time slicing and etc…, some notion of time is needed. On the hardware, there is always a timer responsible for this called the operating system clock timer. It is programmed by Linux PREEMPT-RT to generate an interrupt each tick. Depending on the kernel configuration used at build time the tick frequency can be selected. In the used RTOS, the OS clock is configured to run at 1000Hz, which means that the interrupts occurs every one millisecond. This tick period is considered the scheduling quantum.

The aim of this test is to measure the time needed by the OS to handle this clock tick interrupt. Its results are extremely important as the clock tick interrupt - being on a high level interrupt on the used hardware platform - will bias all other performed measurements.

This test helps also in detecting "hidden" latencies that are not introduced by the clock tick. In such cases, the "hidden" latency will be different and its event time will not be aligned with the RTOS clock tick frequency.

Test method: The way we get the clock tick duration in this test is simple: we create a real-time thread with the highest priority. This thread does a finite loop of the following tasks: starting the measurement by reading the time using the "Start" signal, executing a "busy loop" that does some calculations and stopping the measurement by reading the time again using the "Stop" signal. Having the time before and after the "busy loop" provides the duration needed to finish its job. This "busy loop" is made so that it can run fully in L1 caches and as such it does not introduce latencies by cache misses. In case we run this test on the bare-machine, this "busy loop" will be delayed only by interrupt handlers. As we remove all other interrupt sources, only the clock tick timer interrupt can delay the "busy loop". When the "busy loop" is interrupted, its execution time increases.

When executing this test in a guest OS (VM) running on top of a hypervisor, it can be interrupted or scheduled away by the hypervisor as well, which will result in extra delays.

Figure 2 presents the results of this test on the bare-machine, followed by an explanation. The X-axis indicates the time when a measurement sample is taken with reference to the start of the test. The Y-axis indicates the duration of the "*busy loop*".

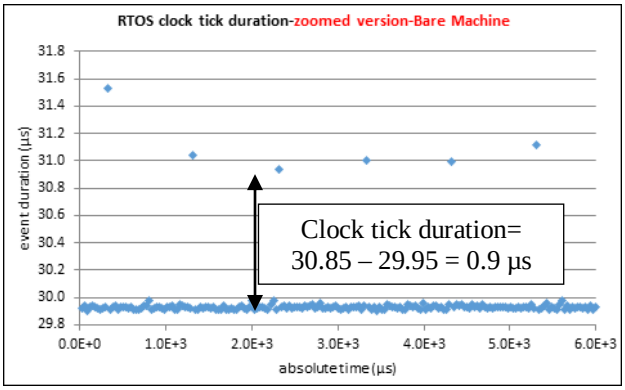

Figure 2: Clock tick processing duration of the bare-machine-zoomed

The lower values (29.95 µs) of Figure 2 present the "*busy loop*" execution durations if no clock tick happens. In case of clock tick interruption, its execution is delayed until the clock interrupt is handled, which is around 31 µs (top values). The difference between the two values is the delay spent handling the tick (executing the handler), which is 0.9 µs.

Remind that the RT Linux kernel clock is configured to generate a tick each 1 ms. This is obvious in Figure 2, which is a zoomed-in version of Figure 3 below.

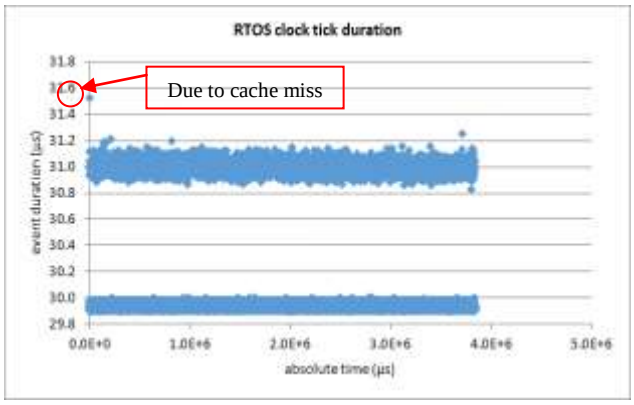

Figure 3: Clock tick processing duration of the bare-machine

Figure 3 represents the test results of 128000 captured samples, in a time frame of 4 seconds (this is limited by the size of the sampling buffers on the used hardware platform). Due to scaling reasons, the samples form a line.
As shown in Figure 3, the "*busy loop*" execution time is 31.53 µs at some periods. Note that the first "clock tick handling duration" (in the circle) took more time (31.53 µs) due to cache miss. Therefore, a clock tick delays any task by 0.9 µs (±0.05 µs) to 1.58 µs (±0.05 µs) µs.

This test detects all the delays that may occur in a system together with its behavior on the short term. To have a long-term view on the hypervisor behavior, we execute a test in the OS (still in the non-virtualized system or bare-machine) for a long duration (more than one hour) where 120 million samples are captured. This test is explained in the following section.

2) Long-term (statistical) clock tick processing duration test

The "clock tick processing duration" test described above detects all the delays that may occur in a system together with its behaviour for a short period. To have a more precise view of the system behaviour, we execute the same test but for a long period. In this test, we use a different sampling method than the previous test due to the sample buffer space limitation. The importance of the figures obtained by "clock tick processing duration" test is to show the exact tracing values and the moments of their occurrence while the figures of "long-term clock tick processing duration" test show their distribution over time and the predictability of the system latencies.

This test is executed 5 times, each time for one hour. The motivation for this (5 times) is to take into consideration all the circumstances that may happen in and around the system, like the room temperature, running the test immediately after the machine start-up, run it after one day of keeping the machine on, etc.

Figure 4 shows the maximum values obtained from each of the 5 runs.

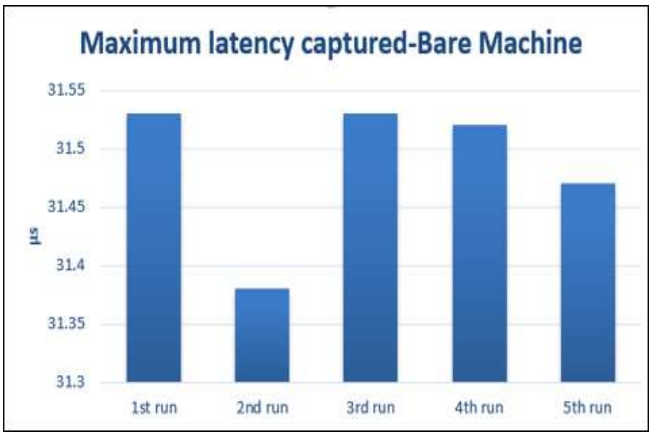

Figure 4: comparing the results of the 5 test runs.

As our concern is about real-time performance, we focus on the test-run where the maximum measurement out of the 5 runs is captured.

Table 1 below shows the results of the run with the highest captured latency.

Test : Statistical clock tick processing duration on bare-machine	Results ±0.05 µs
Test time	1 hour
Samples	120 million
Minimum value	29.95 µs
Maximum value	31.53 µs
Max Tick duration	1.58 µs

Table 1: Statistical clock tick processing duration results for the run with maximum latency

Figure 5 below shows the statistical distribution of the results obtained in the run of our concern.

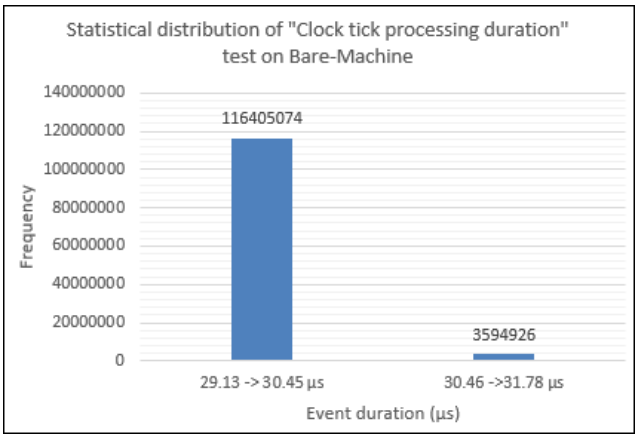

Figure 5: Bare-Machine results for the statistical test with highest latency captured

Figure 5 shows that 97 % of the samples (116405074) are in the interval between 29.13 µs and 30.45 µs. This is logical as the "busy loop" execution time (± 29.95µs) falls in this region. Any samples outside this region are considered delays. We see that 3 % of the samples are between 30.46 µs and 31.78 µs. The maximum value captured in this interval is 31.53µs. Therefore, the maximum overhead detected in the system is 1.58 µs (±0.05 µs) (31.53-29.95).

3) Clock tick processing duration with cache flushing

In the previous test "clock tick processing duration", the clock tick handler is always residing in the cache due to the periodic clock tick interrupt. Also, our test is not fetching a lot of data which could affect the cache contents. This is the best case scenario.

The aim of this test - clock tick processing duration with cache flushing – it to show the worst case duration that may happen in the system due to cache misses. In order to do so, we flush the caches (L1+L2+L3) on a regular interval which in turn causes the clock tick handler to be fetched from the memory whenever a clock tick happens in the system.

For a better understanding of this test setup and outcome, we first provide a zoomed version of the results figure, and then the complete figure.

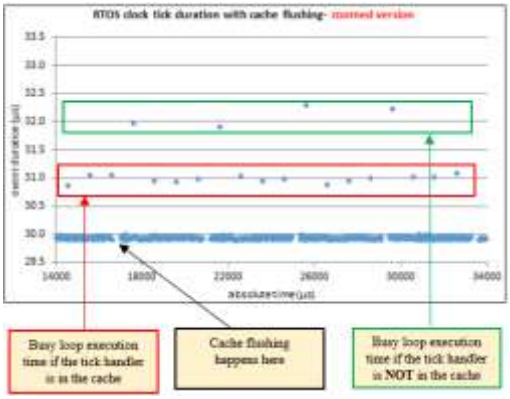

Figure 6: Clock tick processing duration zoomed version of the results on the bare-machine

Figure 6 above shows that this test runs initially for 4ms (X-Axis) where four clock tick interruptions occurred, causing delays in the test execution (the first four samples in the red box). After that, the cache is flushed and the test execution is resumed. When the next clock tick interrupt occurs, the CPU suspend the test until it handles this interrupt. But as the handler is flushed away from the caches, the CPU will fetch it from the RAM again. This fetch costs extra delay in handling the interrupt, which in turn delays the execution of the test (first sample of the green box). The test continues running with the same described procedure.

Figure 7 shows the test results for a longer period.

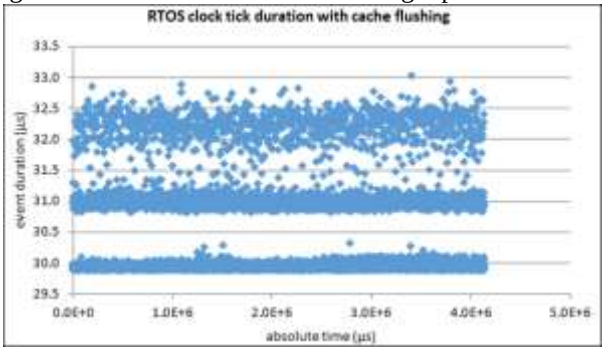

Figure 7: Clock tick processing duration in case of cache flushing

Table 2 below summarizes the results of this test:

Test : Clock tick processing duration with cache flushing	Results ±0.05 µs
Number of captured samples	128000
"Busy loop" time if NO clock interrupt occurs	29.95 µs
"Busy loop" time if clock interrupt occurs	30.95 µs with worst case 33.04 µs
Clock interrupt processing duration	1 µs with worst case of 3.09 µs

Table 2: Clock tick processing duration with cache flushing

4) Statistical clock tick processing duration test with cache flushing

This is the long-duration version of the test "clock tick processing duration with cache flushing".

Again this test is done 5 times, each time for 1 hour (capturing 120 million samples). Figure 8 below shows the maximum values captured in each of the 5 runs.

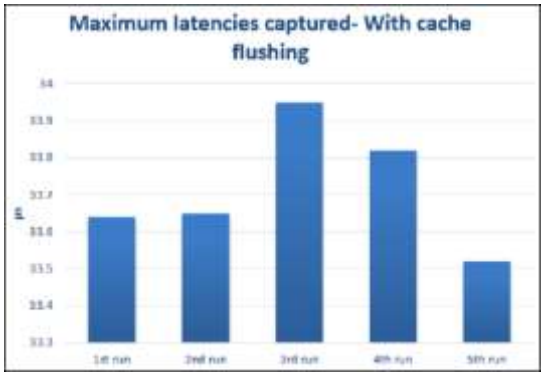

Figure 8: Comparing the maximum values obtained in the 5 runs of statistical clock test with cache flushing

Table 3 summarizes the test results of the run that captured the maximum latency.

Test : Statistical clock test-with flushing	Results ±0.05 µs
Test time	1 hour
Samples	120 million
Minimum value	29.95 µs
Maximum value	33.96 µs
Max Tick duration	4.1 µs

Table 3: The statistical test with cache flushing test run with maximum latency

5) Summary of the four clock tick processing duration tests on the bare machine

Table 4 provides a summary for the maximum ***overheads*** obtained in each of the four clock tick processing duration tests above (executed on non-virtualized systems).

Bare-Machine -- Clock tick duration test-- Maximum overhead		
Test	No cache flushing	With cache flushing
Short-time test	1.58 µs	3.09 µs
Long-time test (statistical test)	1.58 µs ±0.05 µs	4.1 µs

Table 4: Comparison between the four clock tests executed on bare-machine

6) Maximum sustained interrupt latency (or interrupt stress) test

"Interrupt tests" evaluate how the operating system performs when handling interrupts. Low latency interrupt handling is a key system capability of real-time operating systems as RTOSs are typically event driven.

On this platform, we use a PCI device which generates interrupts using an internal timer (thus independent of operating system clock being tested).

This test detects when an interrupt cannot be handled anymore due to the interrupt overload. In other words, it shows a system limit depending on, for example, how long interrupts are masked, how long higher priority interrupts (the clock tick or other) take, and how well the interrupt handling is designed.

It also gives a very optimistic worst case value due to the fact that, because of the high interrupt rate, the amount of spare CPU cycles between the interrupts is limited or nil. Also, depending on the length of the interrupt handler, it might mostly be present in the caches. In a real world environment, the worst case duration will be longer.

In this test, 10 million interrupts are generated at specific interval rates. Our test measures whether the system under test misses any of the generated interrupts. The test is repeated with smaller and smaller intervals until the system under test is no longer capable handling the interrupt load.

Table 5 below show the results of this test:

Interrupt period (±0.05 µs)	#interrupts generated	#interrupts lost
27 µs	10 000 000	4
27.5 µs	10 000 000	1
28 µs	10 000 000	0
28 µs	100 000 000	0

Table 5: Sustained interrupt frequency on bare-machine

The above table shows that the RT Linux OS can handle all the 10 million generated interrupts without missing any one only if the duration between the generated interrupts is 28 µs. Below this value, RT Linux start to miss some interrupts. Further on, the system is tested by generating bursts of higher number of interrupts, which on the long run shows that the guaranteed interrupt duration for Linux on bare-machine is 28 µs (100 million interrupts scenario).

5. TESTING RESULTS ON VIRTUALIZED SYSTEM

RTS hypervisor is tested using the same test metrics as the bare-machine but in several different scenarios or use cases. Below is an explanation of each scenario followed by the test results.

Scenario 1:

The aim of this scenario is to measure the extra overhead introduced in the VM, compared to the bare-machine, due to the insertion of the RTS virtualization layer. In this scenario, we have two virtual machines (VMs) running on top of the hypervisor: a virtualized Windows 7 VM, and a privileged VM with Linux PREEMPT-RT OS. Each VM is assigned to run on one physical CPU. The privileged VM is the UTVM as our tests are performed in it. Windows 7 VM is running in virtualized mode, in ***idle state*** and acts as the interface to connect and control the UTVM.

Figure 9 below is a graphical representation for this scenario.

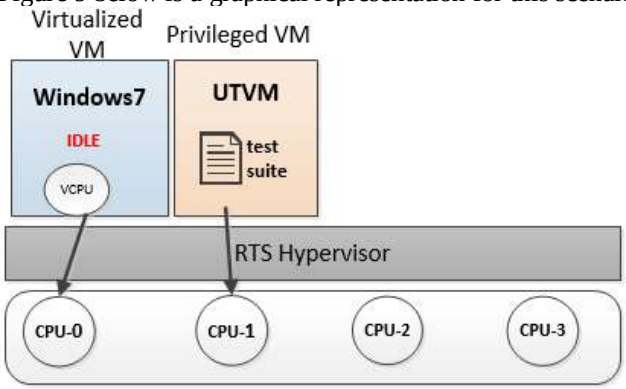

Figure 9: Scenario 1: UTVM and idle Windows VM are running

As all the tests explained in section 4 above will be conducted in every scenario, this will end up in a lot of figures and tables to be shown. Therefore, we decided to provide only a summary of the results.

Figure 10 below shows the results of all the tests, together with a comparison with the results of the bare-machine.

±0.05 µs	Without cache flushing		With cache flushing		Interrupts
	Clock test Short period	Clock test Long period	Clock test Short period	Clock test Long period	Sustained frequency
Bare-Machine	1.58 µs	1.58 µs	3.09 µs	4.1 µs	28 µs
Scenario 1	2.6 µs	5.79 µs	9 µs	9 µs	14.7 µs

Figure 10: Results of the five tests executed in scenario 1

Note: Low values mean better performance.

Figure 10 results show that an extra overhead (the values in the red rectangle), compared with the bare-machine, is captured in the RTOS running atop RTS hypervisor. Normally, there should not be any difference between the values especially that the UTVM is having direct access to the hardware. As we do black box testing, we do not know the exact reason for this extra overhead.

The only strange result is the "maximum sustained interrupt latency" value (in the green rectangle) which is twice better than the bare-machine. In theory, this is not very logical! But this happens due to the fact that RTS-Hypervisor assigns only the IRQs explicitly specified for a specific VM to it. To make it more clearly, we refer back to the bare-machine case where there is only the hardware and an OS running atop of it. In such system, the OS execution is affected by different kind of interrupt sources like the System Management interrupts (SMI) from the BIOS.

The RTS hypervisor, is configured in a way that all the hardware interrupts are directly assigned to the CPU that is servicing the virtualized VM, except the ones that are required by the UTVM. This means that the execution of the UTVM is less affected by external interrupts, and can handle its tasks more rapidly.

This is illustrated and explained in details in Figure 11.

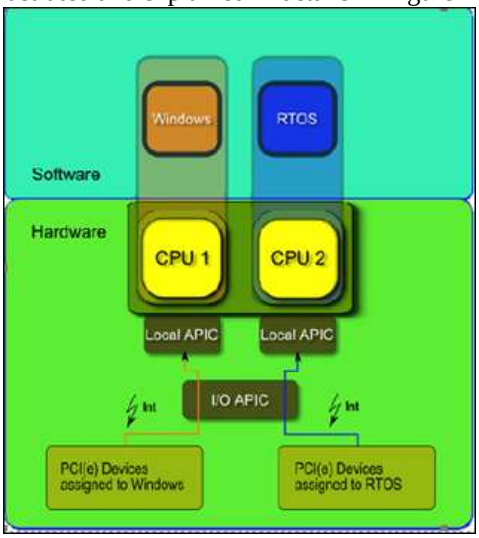

Figure 11: The procedure of configuring interrupts through the RTS hypervisor.

To proof this, we did a test running UTVM solely on top of the hypervisor and assigning all interrupts to it. In this situation the test results were exactly the same as the results of the bare-machine.

Scenario 2:

The aim of this scenario is to detect the effect of CPU caches on the UTVM performance. This scenario is the same as scenario 1 except that the Windows 7 VM is running a Memory-Load stress test. This Memory-Load test unloads all the caches by accessing each cache-line. For this processor, each cache-line consists of 64 bytes of memory. On a cache miss, a cache line is unloaded and replaced by data from the main memory. This happens each time in chunks of cache-line data. By accessing only one word of data each 64 bytes, we stress the memory bus while minimally using the CPU resources. Thus, this generates a worst case stress load towards the central memory bus, which can exceptionally happen in real world systems, for instance when walking through a linked list. For this test we used a loop that flush 9MB of cache so that the complete cache is flushed.

Figure 12 describes this scenario.

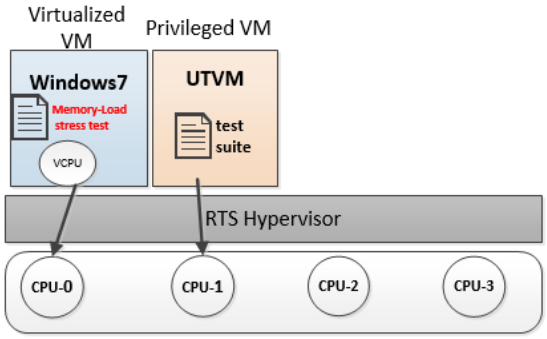

Figure 12: Scenario 2 with Windows VM executing memory stress test

Again, we provide only a summary of all the tests executed within this scenario (Figure 13).

±0.05 µs	Without cache flushing		With cache flushing		Interrupts
	Clock test Short period	Clock test Long period	Clock test Short period	Clock test Long period	Sustained frequency
Bare-Machine	1.58 µs	1.58 µs	3.09 µs	4.1 µs	28 µs
Scenario 1	2.6 µs	5.79 µs	9 µs	9 µs	14.7 µs
Scenario 2	2.4 µs	5.67 µs	5.22 µs	9.4 µs	14.7 µs

Figure 13: Results of the five tests executed in scenario 2

In comparison with scenario 1, scenario 2 results are almost the same which means that one VM of specific workload does not have a big influence on the other VMs.

Scenario 3:

The aim of this scenario is to clarify whether the type of workload in the VMs has any effect on the performance of the UTVM. In this scenario, 4 VMs are running: UTVM, Virtualized Windows VM and 2 other privileged VMs. All the VMs (except UTVM) are doing Memory-Load stress test. Figure 14 presents this scenario.

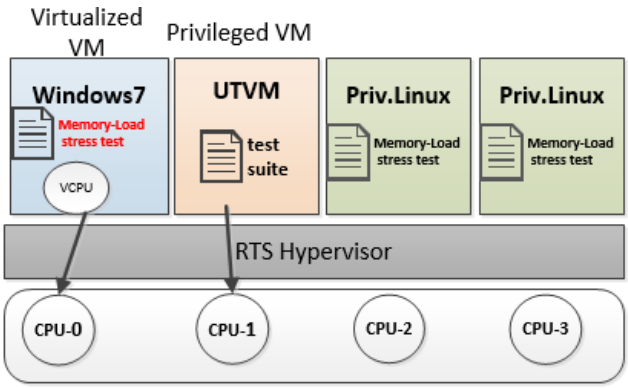

Figure 14: Scenario 3 with three VMs executing memory-load stress test

Figure 15 below is a summary of the test results, compared with the previous two scenarios and the bare-machine.

±0.05 µs	Without cache flushing		With cache flushing		Interrupts
	Clock test Short period	Clock test Long period	Clock test Short period	Clock test Long period	Sustained frequency
Bare-Machine	1.58 µs	1.58 µs	3.09 µs	4.1 µs	28 µs
Scenario 1	2.6 µs	5.79 µs	9 µs	9 µs	14.7 µs
Scenario 2	2.4 µs	5.67 µs	5.22 µs	9.4 µs	14.7 µs
Scenario 3	4.55 µs	10.08 µs	5.6 µs	10.52 µs	14.7 µs

Figure 15: Results of the five tests executed in scenario 3

In all of the four clock tests, scenario 3 has the highest value. This is due to the system memory bus bottleneck. The hardware platform used for this evaluation is a Symmetric Multiprocessor System (SMP) system with four identical cores connected to a single shared main memory using a system bus. These cores have full access to all I/O devices and are treated equally.

The system memory bus or system bus can be used by only one core at a time. If two cores are executing tasks that need to use the system bus at the same time, then one of them will use the bus while the other will be blocked for some time.

As the processor used has 4 cores, when all of these are running at the same time, system bus contention occurs. This

explains the high values obtained in scenario 3. This is as well the worst case scenario that may be expected in any system using this hypervisor on this platform. It has to be remarked that also on a bare metal system bus contention can and will increase execution latencies. Virtualization does not solve this bottleneck. However, using such virtualization solution avoids cache coherency latencies as memory regions are not shared between the different VM.

The sustained interrupt test is not impacted. This is expected due to the nature of the test. The high interrupt rate will keep the interrup thandler cached, so no RAM accesses are needed and thus no bus contention will occur.

6. EXECUTIVE COMPARATIVE SUMMARY:

In this section, we present a graphical comparison between the results of ***long term tests*** in all the scenarios, as the concern is always about worst case situations when dealing with real-time systems.

Clock tick processing duration test for long term - Statistical test- (No cache flushing):

This test measure the RTOS clock tick processing duration. It runs for a long duration (5x1 hour). Figure 16 below shows the maximum **overhead** obtained in each scenario.

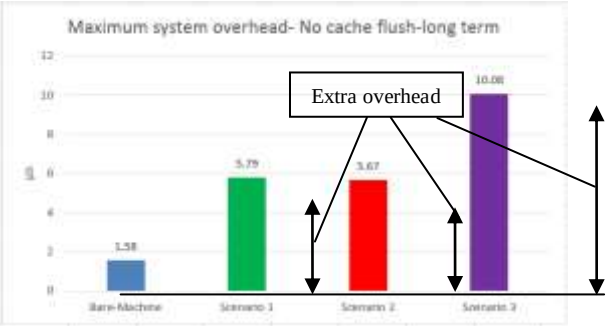

Figure 16: Comparison between the maximum overheads captured in each scenario for the long term clock test.

Clock tick processing duration test for long term - Statistical test- (With cache flushing):

This test is the same as the "clock tick processing duration" test except that the cache is flushed at periodic durations. It is executed during a long period. Figure 17 shows the results of this test in each scenario.

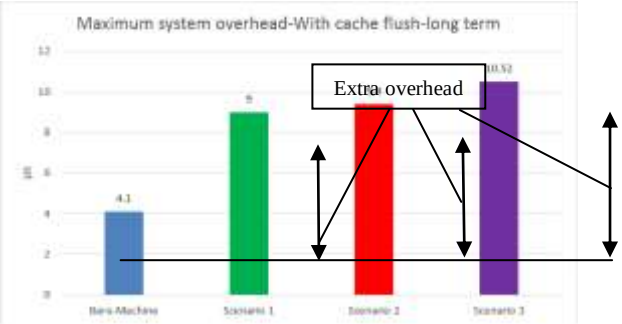

Figure 17: Comparison between the maximum overheads captured in each scenario for the long term cache flushing clock test.

Sustained interrupt frequency:

This test measures the probability that an interrupt might be missed. Figure 18 compares the values of the results in all scenarios.

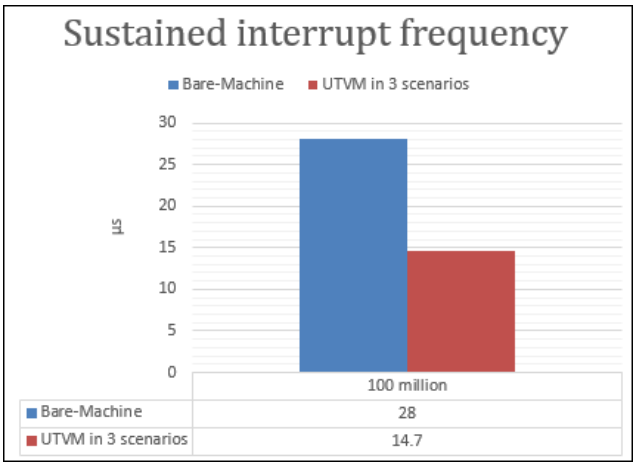

Figure 18: comparison between the sustained interrupt duration for all scenarios

Although there are no scheduling decisions in the RTS hypervisor and moreover it allows the RTOS to have direct access to the hardware, an overhead is detected. The "clock tests" of Figure 16 show that on average 4 µs overhead is captured in the virtualized system in all the scenarios compared to the non-virtualized system.

The system caches can affect the RTOS performance by increasing the system overhead of near 3 µs, which is visible by comparing the corresponding scenarios of Figure 16 and Figure 17.

The architecture of the SMP hardware also plays a role in performance degradation especially with memory-stress workload, which causes system memory bus bottleneck. For instance, with a 4-CPU hardware, the performance is degraded by a factor 2 which can be seen in scenario 3 in all "clock tests".

Even with all these increases, the worst case latencies in a RTOS atop RTS are still bounded and the system remain predictable.

7. CONCLUSION

For our experimental evaluation of RTS, we tested the real-time performance of a privileged VM running Linux PREEMPT-RT. The results of the short-term tests (only a couple of seconds) show that the performance of the RTOS running as guest in a privileged VM is almost the same as running RT Linux directly on a bare-machine (non-virtualized). However, the results of the sustained tests (statistical tests for long duration) show that an overhead of about 4µs is added to the RT VM performance. This overhead can increase up to 8 µs in case of system memory bus bottleneck.

Even though an overhead is detected in the RT VM, it is bounded and the RTOS performance remains predictable, a requirement in hard real-time systems.

Moreover, RTS shows a great behaviour when dealing with interrupts due to its mechanism of assigning interrupts to the VMs. Most of the interrupts (including SMI) are handled by the virtualized VM except the ones that are explicitly specified to be handled by the RT-VM. This means that less interruption happens in the RT-VM which results in shorter worst case latencies than on a bare-metal system.

Therefore, RTS is a highly recommended virtualization product for hard and soft embedded real-time systems.

8. Acknowledgment

We would like to thank RTS Company for making its product available for testing, and for their support during the evaluation process.

9. References

[1]. J. Zhang, K. Chen, B. Zuo, R. Ma, Y. Dong and H. Guan, "Performance analysis towards a KVM-Based embedded real-time virtualization architecture," in 5th international conference on Computer Sciences and Convergence Information Technology (ICCIT), Seoul, 2010.

[2]. Z. Gu and Q. Zhao, "A State-of-the-Art Survey on Real-Time Issues in Embedded Systems Virtualization," Journal of Software Engineering and Applications, 2012.

[3]. H. Gernot, "Virtualizing embedded systems - why bother?," in 48th ACM/EDAC/IEEE Design Automation Conference (DAC), New York, 2011.

[4]. M. Lee, A. Krishnakumar, P. Krishnan, N. Singh and S. Yajnik, "Supporting soft real-time tasks in the xen hypervisor," in Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments , New York, 2010.

[5]. Real Time Systems GmbH, "Real-Time Virtualization, Embedded Hypervisor and Real-Time Windows Solutions for Windows Real-Time Applications," [Online]. Available: http://www.real-time-systems.com/.

[6]. Luc Perneel, Hasan Fayyad-Kazan and Martin Timmerman, "Android and Real-Time Applications: Take Care!," Journal of Emerging Trends in Computing and Information Sciences, no. 2013.

[7]. Hasan Fayyad-Kazan, Luc Perneel and Martin Timmerman, "Linux PREEMPT-RT vs. commercial RTOSs: how big is the performance gap?," GSTF Journal of Computing, 2013.

[8]. K. Adams and A. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating System, New York, NY, 2006.

[9]. P. Braham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt and A. WarField, "Xen and the art of virtualization," in Proceedings of the nineteenth ACM symposium on Operating systems principles, New York, NY, 2003.

[10]. F. Bellard, "QEMU, a fast and portable dynamic translator," in Proceedings of the USENIX 2005 Annual Technical Conference, CA, 2005.

[11]. J. Smith and R. Nair, Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design, San Francisco: Morgan Kaufmann, 2005.

[12]. R. Uhlig, G. Neiger, D. Rodgers, F. Santoni, F. Martins, A. Andersons, S. Bennett, A. Kagi and L. Smith, "Intel virtualization Technology," Computer 38, 2005.

[13]. J. Watson, "VirtualBox: bits and bytes masquerading as machines," Linux Journal, 2008.

[14]. D. Ferstay, "Fast Secure Virtualization for the ARM Platform," Master's thesis, The University of British Columbia, Faculty of Graduate Studies, 2006.

[15]. G. Heiser, "The Motorola Evoke QA4—A Case Study in Mobile Virtualization," Technology White Paper, Open Kernel Labs, 2009.

[16]. K. Dong-Guen, L. Sang-Min and S. Dong-Ryeol, "Design of the Operating System Virtualization on L4 Microkernel," in Fourth International Conference on Networked Computing and Advanced Information Management, Gyeongju, 2008.

[17]. K. Sandstrom, A. Vulgarakis, M. Lindgren and T. Nolte, "Virtualization Technologies in Embedded Real-Time Systems," in 18th Conference on Emerging Technologies & Factory Automation (ETFA) , Cagliari, 2013.

[18]. H. Fayyad-Kazan, Benchmarking Virtualization solutions for business and embedded systems, PhD Thesis, Brussels: University Press, ISBN: 978-9-4619721-5-6, 2014.