The Need For Portable Benchmark to Evaluate The Performance of Real Time Operating Systems

Shreesha Rao P
Department of Computer Science and Engineering
SJBIT
Bangalore, India
p.shreesha.rao@gmail.com

Chandan K N
Department of Computer Science and Engineering
SJBIT
Bangalore, India
chandankn.321@gmail.com

***Abstract*— The process of performance comparison of two or more systems by measurements is called benchmarking, and the workloads used in measurements are called benchmarks. Benchmarking of computer systems is an important sometimes tedious task that gives insights into the performance of a system, exposes flaws, and allows for comparison between systems. Thus from the Application programmer's perspective, it helps them choose an Real Time Operating System which suits their application the best and from the OS programmer's perspective, it helps them fine tune various aspects of Real Time Operating Systems.**

Keywords*— *Embedded Systems, Kernel, Real Time Operating Systems (RTOS), RTOS benchmarking, Performance test and Performance evaluation.

I. INTRODUCTION

Real-time systems can be said to be systems that have to perform tasks timely. An aircraft, a car, a mobile phone and a radio base station are examples of machines and devices with Real-Time properties. A Real-Time system has one or more tasks it has to perform, the tasks can be of different priority and may have deadlines which must be met. It's the task of the Real Time Operating System to handle priorities and switch between tasks. The order in which the tasks are executing are decided by a scheduling algorithm. System performance. A Real Time Operating System provides a set of system calls to the developer of a real-time system. Support for semaphores, timers, scheduling and inter process communication such as message passing are common services. The use of such services facilitates the development of a real-time system but imposes an overhead to the application. This overhead depends on the implementation of the real-time operating system. It may be useful or even necessary for an engineer to be able to quantify this imposed overhead on the real-time system to verify that task deadlines are met. A Real Time Operating System is different from a desktop or server operating system by usually being much smaller and focus on deterministic and timely behavior. Real Time Operating System kernels are based on microkernel architecture, which means they are small and most services and drivers are executed outside kernel space.

A. RTOS benchmarking

There are different approaches towards RTOS benchmarking: based on applications or based on the most frequently used system services (fine-grained benchmarking) [1] As there are various types of applications with each having very different requirements, benchmarking against any generic applications will not be reflective of the RTOS strengths and weaknesses. There are various research publications related to benchmarking method based on frequently used system services. In [2] the Rhealstone benchmark is proposed with the following measurement: task switch time, preemption time, interrupts latency time, semaphore shuffling time, deadlock breaking time, and datagram throughput time. Rhealstone benchmark is not suitable for several reasons. Firstly, very few RTOSes are capable of breaking deadlock (which we will see later in the RTOSes survey). Datagram throughput time is based on message passing by copying to a memory area managed by the OS. However, not all RTOSes use the same concept for message passing. Some RTOSes pass messages by passing only the memory pointer, and hence there is no need to use the special memory area managed by the OS. This approach is also more suitable for small microcontrollers because there is no extra memory for OS internal use. Interrupt latency time as defined by Rhealstone is purely dependent on the CPU architecture and is not determined by the RTOS. Rhealstone, in general, are "somewhat adhoc", and do not cover other situations commonly found in real-time applications [1].

In [1], some metrics are proposed (based on frequently used system services): inter-task synchronization and resource sharing, and inter-task data transfer (message passing). Inter-task data transfer, as explained previously, is also based on data copying into a memory area managed by the OS, similar to the "datagram throughput time" in Rhealstone benchmark. In the test for "response to external event (interrupt)", the interrupt handler wakes up another task via a semaphore. Using a semaphore in this case does not seem to be the best approach. Waking up the task directly by using system service call (such as sleep/wakeup service call) instead of going through a semaphore is a better approach to reduce the overhead delay.
In [4], the metrics proposed are (based on frequently used system services): tests for measuring the duration of message transfer and the communication through a pipe, tests for measuring the speed of task synchronization through proxy and signal, and tests for measuring the duration of task switching.

II. RTOS FEATURES

A. Criteria for comparison

The objective of this section is to investigate RTOSes available (open-source,commercial, and research) and determine those that are suitable. Information is mainly based on documentations and APIs available on websites. These RTOSes are: µITRON, µTKernel, µC-OS/II, EmbOS, FreeRTOS, Salvo, TinyOS, SharcOS, XMK OS, Echidna, eCOS, Erika, Hartik, KeilOS and PortOS.

As described in [5], criteria used for selecting an RTOS includes the following: language support, tool compatibility, system service APIs, memory footprint (ROM and RAM usage), performance, device drivers, OS-awareness debugging, tools,technical support, source/object code distribution, licensing scheme and company reputation. Similarly criteria mentioned in [6] are: configuration, API richness, documentation and support and and tools support.

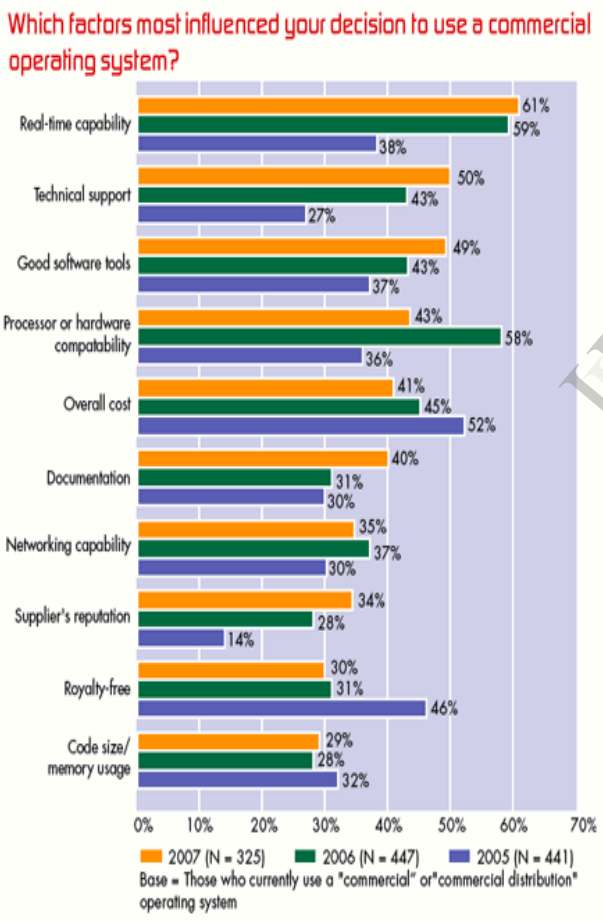

Figure 1: Influential factors in operating system selection .

Design objective: The origins of the RTOSes being surveyed are different from one another, as some are open-source, some are personal hobby-based ,and some are commercial. It is important to understand the history and the background motivation that led to the creation of each RTOS. A personal hobby-based RTOS would be less likely to be as stable compared to a popular open-source, or to a commercial RTOS.

Scheduling scheme: RTOS scheduling approach will be investigated to determine whether preemptive, cooperative or other scheduling scheme is used.

Real-time capability and performance: Real-time capability is generally considered as a system characteristic to describe whether the system is able to meet the timing deadline. Using an RTOS in the system takes up CPU cycles; however the RTOS must not have indeterminist behaviors. The amount of CPU cycles and time consumed by the RTOS for any service call should be measurable and of low or acceptable values to the system designers. Real-time capability and performance information are not available for some RTOSes. Even if these information are available, they might be based on different hardware platforms.

Memory footprint: Besides CPU cycles, an RTOS also occupies additional ROM and RAM spaces. This could lead to larger ROM and RAM sizes for the entire system. There is always a tradeoff between memory footprint and the functionalities required from the RTOS. To have more robust and reliable APIs, probably more lines of code are needed. On the other hand, basic and simple APIs will require only minimum amount of code.

Language support: Programming language supported by the RTOS.

System call/API richness: This criterion determines how comprehensive the RTOS APIs are as compared to the rest of the RTOSes. The total number of system calls for each RTOS will be counted.

OS-awareness debugging support: This criterion determines if the RTOS is being supported by any of the Integrated Development Environment (IDE). OS-awareness debugging [3] will ease the development work as users can use these RTOS internal information .

License type: This is to investigate how the RTOS is distributed: free or fee-based for different purposes such as educational or commercial.

Documentation: This criterion will focus on what type of documentations is available for the RTOS (detail APIs, simple tutorial, book or specification).

III. PERFORMANCE BENCHMARKING

A. Benchmarking methods

The following RTOSes will be benchmarked: µITRON, µTKernel, µC-OS/II and EmbOS. As discussed in the previous section.These four RTOSes are made

available on the same platform for the purpose of benchmarking:

The Renesas M16C/62P starter kit with the HEW IDE together with the NC30 tool chain [7] is used. For execution time measurements, oscilloscope and logic analyzer have been used in combination with IO port toggling to achieve the best accuracy (in terms of micro-seconds).

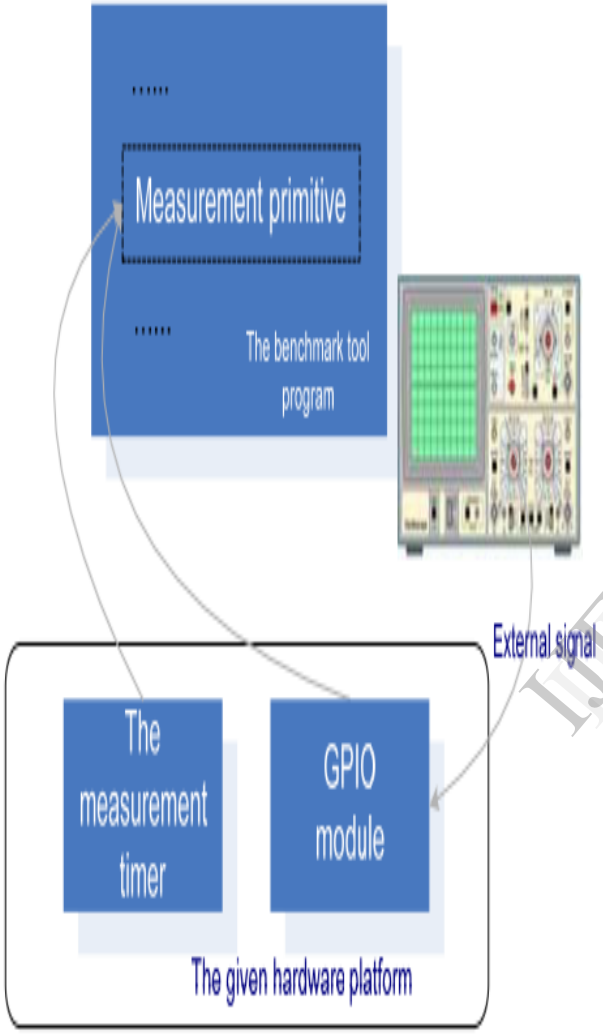

Figure 2: Oscilloscope based hardware

The Renesas M16C/62P has a 16-bit CISC (complex instruction set computer) architecture CPU with a total of 91 instructions available. Most instructions take 2 to 3 clock cycles to complete. The MCU is designed with a 4-stage instruction queue buffer which is similar to a simplified pipeline often used in larger 32-bit processor

A. *Benchmarking criteria*

The proposed benchmarking criteria in this section are aimed to be simple and easy to port to different platforms. For each criterion, execution time measurement together with memory footprint (ROM and RAM) will be collected.

(a) *Task switch time*

Task switch time is the time taken by the RTOS to transfer the current execution context from one task to another task.

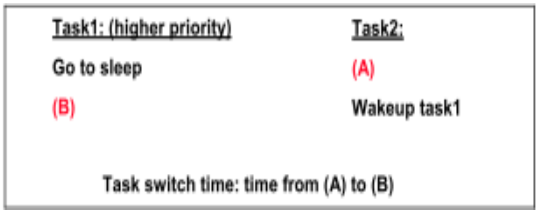

Figure 3: Task switch time measurement

	µC-OS/II	µTKernel	EmbOS	µITRON
Pass message API	OSTaskSuspend()	tk_slp_tsk()	OS_Suspend()	slp_tsk()
Retrieve message API	OSTaskResume()	tk_wup_tsk()	OS_Resume()	wup_tsk()

Table 1: API used for task switch time

There are two tasks: Task1 and Task2 with Task1 having higher priority. At the beginning, Task1 is first executed, and it will go into sleep/inactive state. The execution context is then switched over to Task2. Task2 will wake up/make active Task1, and right after waking up, the execution context is switched over to Task1 because it has higher priority. Different RTOSes use different terms to describe sleep/inactive and ready/active states (such as µC-OS/II and EmbOS use the term suspend, resume while µITRON and µTKernel use the term sleep/wakeup).

(b) *Get/Release semaphore time*

Semaphore is commonly used for synchronization primitive in RTOS. For semaphore benchmarking, the time taken by get and release semaphore service call will be measured, and the time required to pass the semaphore from one task to another task will also be measured.

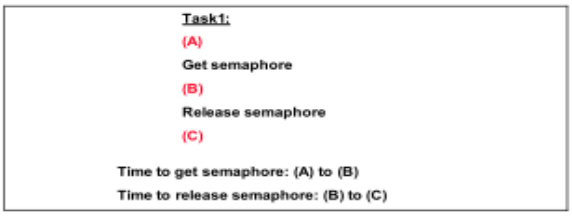

Figure 4: Get/Release semaphore time.

	µC-OS/II	µTKernel	EmbOS	µITRON
Get semaphore API	OSSemPend()	tk_wai_sem()	OS_WaitCSema()	wai_sem()
Release semaphore API	OSSemPost()	tk_sig_sem()	OS_SignalCSema()	signal_sem()

Table 2: API used for semaphore benchmark.

(c) *Semaphore passing time*
To measure the performance of semaphore passing the following measurement is used.

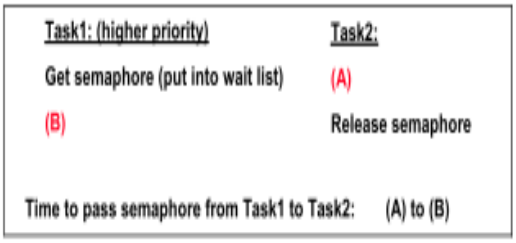

Figure 5: Semaphore passing time measurement.

There are two tasks: Task1 and Task2 with Task1 having higher priority, and a binary semaphore (initialized to 0). At the beginning, Task1 is first executed and it tries to get the semaphore. Since the semaphore value is 0, Task1 will be put into a sleep/inactive state, waiting for the semaphore to be released. The current execution context will then be switched to Task2 which will release the semaphore. The semaphore, once released, will wake up Task1, and the execution context will be switched over to Task1.

(d) *Pass/Receive message time*
Besides semaphore, message passing has become more and more popular for synchronization purposes. In this message passing mechanism based on memory pointer passing is used, i.e. Not the copying of message into an internal RTOS area because not all RTOS support this approach.

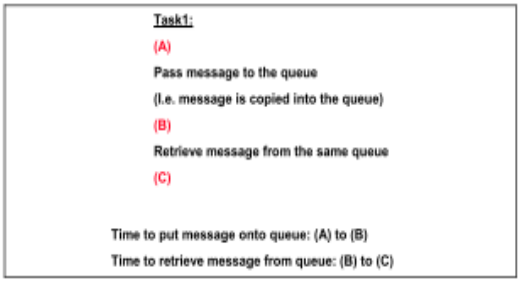

Figure 6: Pass/Retrieve message time.

	µC-OS/II	µTKernel	EmbOS	µITRON
Pass message API	OSQPost()	tk_snd_mbx()	OS_Q_Put()	snd_mbx()
Retrieve message API	OSQPend()	tk_rcv_mbx()	OS_Q_GetPtr()	rcv_mbx()

Table 3: APIs for message passing benchmark

(e) *Fixed-size memory acquire/release time*
In RTOS, only fixed-size dynamic memory allocation should be used

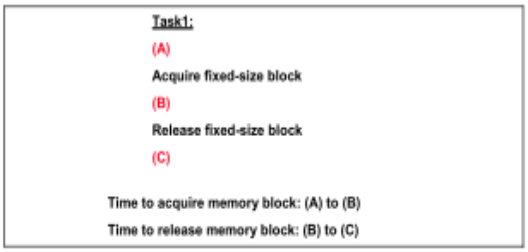

Figure 7: Acquire/Release time measurement.

(f) *Task activation from within interrupt handler time*
An RTOS has to deal with external interrupts that may be asserted at any time. Execution of interrupt handler is normally kept as short as possible to avoid affecting the system response. In the case where long processing is required, the handler can activate another task that will do the necessary processing. The time from when the interrupt handler resumes the task till the time when the task is executed is crucial to the system design.

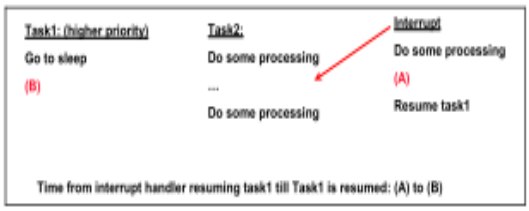

Figure 8: Task activation from interrupt handler time measurement.

	µC-OS/II	µTKernel	EmbOS	µITRON
Go to sleep API	OSTaskSuspend()	tk_slp_tsk()	OS_Suspend()	slp_tsk()
Resume from interrupt API	OSTaskResume()	tk_wup_tsk()	OS_Resume()	iwup_tsk()

Table 4: APIs for task activation from within interrupt handler benchmark.

B. Benchmarking results

(1) Memory footprint

For each criterion, the benchmarking code is compiled, and the ROM and RAM usage can be obtained from the toolchain report. By averaging the ROM information across all the test criteria, the average ROM size can be obtained. Figure 9 shows the code sizes for the 4 RTOSes when running the 7 benchmarks. µTKernel can be seen to have a larger code size.

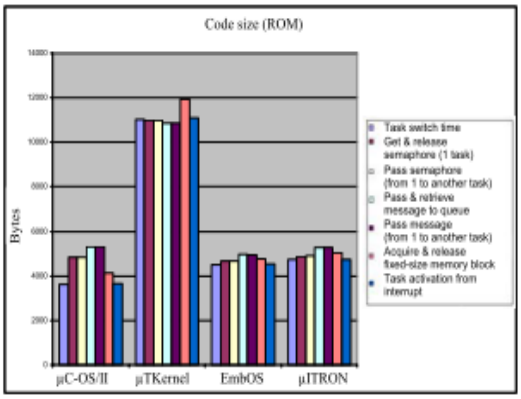

Figure 9: Code size comparison for 4 RTOSes.

Similar to the code size comparison, Figure 10 shows the RAM information for the 4 RTOSes. µTKernel and µITRON can be seen to have relatively lower RAM usage, while µC-OS/II and EmbOS are slightly higher. Depending on the requirement of each benchmark, we set the number of tasks, stack size and number of RTOS objects (e.g. semaphore, event flags) to be the same for all RTOSes. The amount of RAM differences among the RTOSes range between 7-10 bytes, which might be due to internal implementations or due to method of designing the APIs.

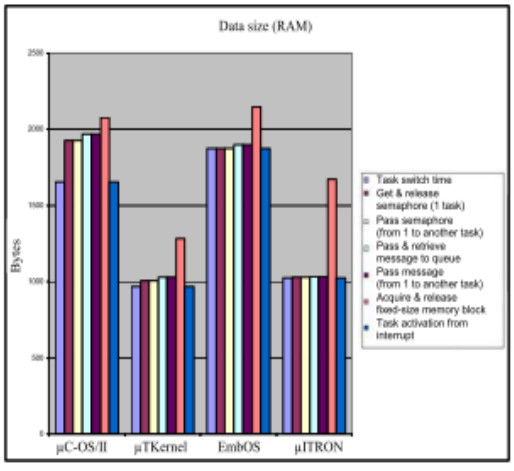

Figure 10: Data size comparison for 4 RTOSes

(2) Execution time

Figure 11 shows the measurement of execution time among the RTOSes for different benchmark criteria. As the only variation in the system is timer interrupt (for OS tick), each benchmark was executed at least twice to ensure consistent results. Nevertheless for all benchmarks, running once is enough to yield the correct measurement. For the task activation from within interrupt handler benchmark, there will be another variation which is an external interrupt (besides the timer interrupt for OS tick). When performing this benchmark, the external interrupt may or may not be asserted during the OS critical section (the duration which OS disable interrupts). If it is asserted during the critical section, the response time of the OS will be slightly longer. Hence this measurement maynot include the worst case scenario. µTKernel is shown to have the lowest task switching time, followed by µITRON, µC-OS/II and EmbOS.

On the other hand, µC-OS/II semaphore acquires is the fastest.The fastest inter-task semaphore passing is achieved by µITRON, while µC-OS/II and µTKernel have better message passing and message retrieval time as compared to µITRON and EmbOS. As far as fixed-size memory is concerned, µC-OS/II has the best execution time, followed by EmbOS, µITRON and µTKernel. Finally, µTKernel has the best performance time for task activation from interrupthandler, followed by µC-OS/II, µITRON and EmbOS. With the benchmarking results shown above, each RTOS stands out to have its own strengths and weaknesses. As far as open-source RTOS is concerned, for a very small and compact ROM size RTOS, µC-OS/II can be used.

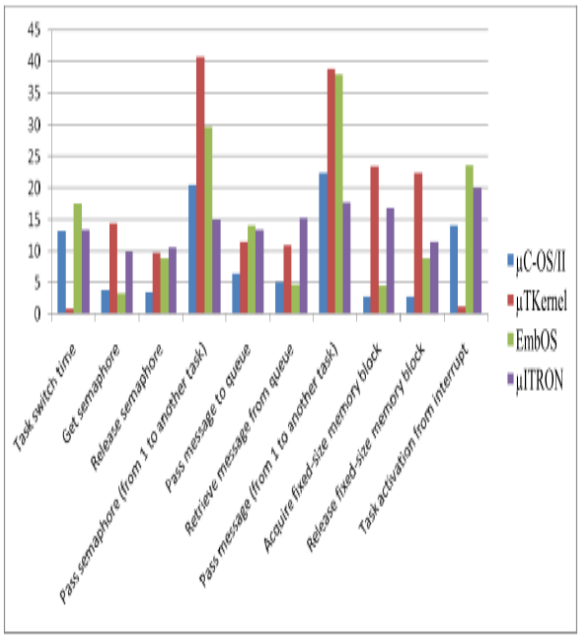

Figure 11: Execution time benchmark for four RTOSes

CONCLUSION

Though there are many benchmarks to evaluate the performance of Real Time Operating Systems, it has been found that there is no single benchmarking tool that can be considered the standard tool or method of benchmarking Real-Time Operating Systems. It's common for Real-Time Operating System vendors to publish a subset of the Rhealstone metrics, but information about the hardware setup and the system calls used is often missing, this limits the usefulness of such figures and makes it hard to draw any real conclusions from them. There is a need to develop a portable benchmark tool that needs to have similar and fair implementation for different Real Time Operating systems. A portable benchmark tool's implementation should separate hardware and OS dependant code from the benchmarking code, and try not to make any weak assumptions about the operating system and the hardware it will execute on. It should be modular enough to facilitate benchmarking of special hardware features or algorithms to be implemented by the user according to the requirements of a special application.

REFERENCES

[1] A. Martinez, J. F. Conde, A. Vina, "A comprehensive approach in evaluation of the performance for modern Real Time Operating Systems." Proceedings of the 22nd EUROMICRO Conference, pp. 61, 1996.

[2] R. P. Kar, K. Porter, "Rhealstone: A Real Time benchmarking proposal", Dr. Dobb's Journal, Feb. 1989.

[3] J. Ganssle, "The challenges of Real Time programming",Embedded System Programming magazine, vol. 11, pp. 20–26, Jul. 1997.

[4] K. M. Sacha, "Measuring the real-time operating system performance", Seventh Euromicro workshop on Real-time systems proceedings, Odense, Denmark, pp. 34–40, Jun. 1995.

[5] G. Hawley, "Selecting a RTOS" Embedded System Design Magazine 1999.

[6] M.Timerman,L.Perneel, "Understand RTOS market", Dedicated Systems RTOS Evaluation project report, Sep. 2005.

[7] Renesas Technology.,"Renesas high performance, Embedded workshop http://www.renesas.com/fmwk .jsp?cnt=ide_hew_tools_product_landing.jsp&fp=/products/tools/ide/ide_hew/, 2007

[8] "RTX Real-Time Kernel," KEIL, http://www.kiel.c Om/rl-arm/kernel.asp

[9] K. Yu and N. Audsley, "A Generic and Accurate RTOS-centric Embedded System Modelling and Simulation Framework,UK Embedded Forum

[10] K. Yu and N. Audsley, "Combining Behavioural Real-time Software Modelling with the OSCI TLM-2.0 Communication Standard," in 7th International Conference on Embedded Software and Systems, (ICESS '10), 2010.