Measurement of OS Services and Its Application to Performance Modeling and Analysis of Integrated Embedded Software*

Shige Wang, Sharath Kodase, Kang G. Shin, and Daniel L. Kiskis
Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122
email:{wangsg,skodase,kgshin,dlk}@eecs.umich.edu

Abstract

Performance analyses for embedded software construction with existing components require knowledge of performance characteristics of both application software and operating system (OS) services, especially those services that are critical for real-time applications. Since end users normally do not control the structure and implementation of OS services, but have to use them to meet the system-level performance constraints, it is essential and critical to characterize the performance of OS services with measurements. As such measurements are taken for performance analysis, not for comparison, the measurement methods should be different from those traditionally used for comparison. In this paper, we present an end-to-end method for measuring the performance of timing and scheduling services in selected real-time OSs for the performance modeling and analysis. The proposed method takes the factors of both OS implementations and application configurations into account to obtain the measured performance close to what applications will experience at runtime. The results have shown that the performance characteristics of OS services can be measured without instrumenting the kernel source code, and hence, can be reused for the analysis of a family of applications.

1 Introduction

Embedded SoftWare (ESW) has usually been implemented as an integration of existing domain-specific control algorithms and device drivers that interact with the external physical world. Such ESW normally runs on resource-limited hardware and is subject to stringent timing constraints to meet the underlying control application requirements. Performance analysis is, therefore, essential and critical during the design and integration of ESW to guarantee that the application timing constraints will be met at runtime. As real-time operating systems (RTOSs) are maturing, most ESW tends to use the existing RTOSs for runtime resource management. An RTOS can be viewed as a set of functional components, each of which provides a service to the application. Most performance analysis techniques require knowledge of the performance characteristics of RTOS services and application-level performance characteristics for design phase performance analysis. Although RTOS vendors provide some product performance characteristics such as throughput, interrupt latencies, and context switch time, such information is usually insufficient for ESW performance analysis. Therefore, it is common for system designers to measure the RTOS performance to meet their domain modeling and analysis requirements. Since most RTOSs are released only as binaries, the measurements are expected to be done without source code. Furthermore, the measurements should be made for each service since "service" is a basic OS unit used in ESW integration. A traditional measurement method meeting these requirements is benchmarking. However, benchmarks, in general, only produce a single, statically configured workload on the system. Thus, a different benchmark must be run for each application configuration, resulting in costly duplication of measurement effort.

In this paper, we present a new approach for measurement and application of the performance characteristics of RTOS services for design-time performance analysis of ESW development and integration. Our method is based on end-to-end measurements using a combination of microbenchmarks and synthetic workloads. End-to-end (e2e) measurements obtain performance information as an external observer, and therefore, minimize the interference

*The work reported in this paper was supported in part by DARPA under the US AFRL contract F30602-01-02-0527.

0-7695-1739-0/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

during measurements and provide the performance results close to what the application will experience at runtime. The e2e method can also uncover inter-service dependencies and performance metrics without RTOS source code, and can be applied to any system-level service like middleware and some subsystem services. Microbenchmarks are an effective method for measuring individual and independent OS-level services without instrumenting the kernel. However, as are typically used, they exercise the system in a limited way that is not necessarily representative of an actual application workload. We can derive more realistic performance metrics by coupling the microbenchmarks with representative, domain-specific synthetic workloads. Synthetic workloads allow the measurements to be taken under conditions close to the real applications with representative resource usage and interaction patterns. Through such synthetic workloads, our measurements can cover most frequently-used application configurations and interaction patterns so that the results can be reused to analyze a family of applications.

In this paper, we demonstrate this technique by measuring the performance of two fundamental RTOS services: timing and scheduling services. The measurements are made on selected RTOSs while varying workloads so that we may obtain the performance characteristics of these services under different platform and application configurations. Our measurement technique makes it possible to use the evaluation results for performance analysis of ESW running on the same targets used for the measurement.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 describes a hierarchical performance model with reusable components, where the performance measurements with the proposed method will be used. Section 4 presents the measurement methodology and the design of experiments for measuring timing and scheduling services, including the measurement environments, measured parameters, measurement tools, and test programs. Section 5 presents our measurement results and the corresponding analysis of measured timing and scheduling services. The paper concludes with Section 6.

2 Related Work

In recent years, many models have been proposed for OS and integrated ESW performance analysis [1–3, 7]. All these models suggested a modeling hierarchy with the OS performance model as a separate layer between hardware and the application, and required the measurement of OS services for model construction and performance analysis. There have also been numerous general techniques for measuring the different aspects of computer systems [5, 9], and bechmarks for OS-level measurements, such as Rhealstone [6], Hartstone [15], *lmbench* [10], and *hbench-OS* [3]. Although these techniques are useful in measuring the performance of individual operations and determining performance bottlenecks, the effects of application structures and interactions — which may make significant performance differences of ESW — are ignored. The information measured with these benchmarks is limited and inaccurate for ESW performance analysis.

Measurements that include the effects of applications can be made with a synthetic workload. Methods of generating synthetic workloads include Hartstone [15], a synthetic workload specification language (SWSL) and generator [8], and DynBench [12]. All these techniques considered application properties, but the performance metrics and frameworks were not defined for RTOS service measurements. Some *ad hoc* methods have also been used for performance measurements of OS services [4, 11, 13] with domain-specific applications, although the measurements were usually restricted to a specific configuration used for the purpose of comparing different products.

Our measurement method focuses on how to obtain information on the performance of various RTOS services under representative application configurations, that is necessary for the analysis of ESW design and integration. This is different from the measurements targeted for product comparisons in that (a) the measurements should be as non-intrusive as possible, (b) the measured information should be reusable for a family of applications, and (c) measurements should encompass all possible configurations of the service in which it can be used.

3 Performance Modeling and Construction Methodology

Our goal is to support ESW analysis for design and integration by providing reusable performance data for RTOS services. The performance measurement and performance model construction for such analysis vary from one application domain to another. The application domain of interest is embedded controls, such as machine control, avionics mission computing, and automotive vehicle control. In such a domain, the ESW is modeled as a set of functional components which interact with one another and also with the environment. In a component-based software architecture, the components which comprise a system can be designed and implemented in a modular manner so they may be reused for a family of applications, as described in [14]. Constructing ESW is then a process of integrating these components. The performance model should, therefore, be constructed based on the performance of individual components and their interactions on a given platform. Performance analysis at design time will be greatly improved if the performance information for each component is included in the component model, as shown in Figure 1.

0-7695-1739-0/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

In this model, the performance of each behavior of a component is modeled with two types of information: *performance requirements* and *performance characteristics*. These are used as input to the analysis process, and are referred to as performance parameters for the component. The requirements normally come from higher-level application constraints. Their primary use in the analysis process is to determine if individual and overall system requirements are met for a given software configuration. Requirements are generally reusable for analyzing a given application in different execution environments. The requirements also help determine the types of performance metrics which are associated with the component's behavior. These metrics form the performance characteristics in our model. They may be directly measured or computed from the measured information. The characteristics depend on the implementation of the behavior and its execution environment. They can be reused to analyze different applications in the same environment.

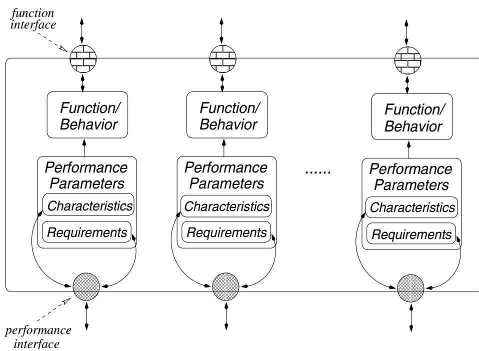

Figure 1. The component model with performance information.

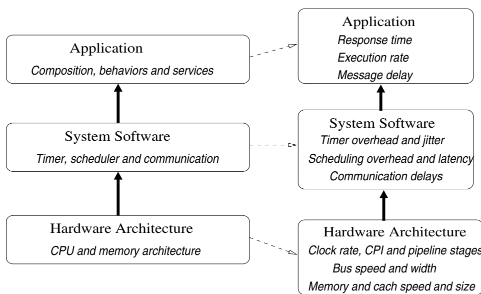

Figure 2. Hierarchical, analytical performance model.

As the ESW is constructed hierarchically by integrating components, the performance information is immediately available for performance model construction and analysis. The performance model is constructed hierarchically, reflecting realization layers [1], as shown in Figure 2. In this paper, we treat all software components running on top of an OS as applications, although a finer-grained model with more layers representing other system software, such as middleware, can be constructed in the same way.

The proposed e2e measurement methodology is used to obtain the performance characteristics. The e2e measurement is a runtime, sampling-based method, which records the start and end times of an activity of interest. The performance characteristics of a service are defined as a set performance metrics for the measurements.

The experiments for measuring service performance include "representative" service requests and application configurations. The measured values will be collected at the application level. Since the existence of design patterns in a given domain usually leads to a small number of likely application configurations, synthetic workloads can be used to generate such representative application configurations with good coverage. To consider the effects of hardware architecture, our measurement will be applied directly to each combination of hardware and RTOS. Although the measured results of service performance will be hardware-specific, techniques [3, 10] exist to break the dependencies further.

4 Design of Experiments to Measure Timing and Scheduling Services

We view each service, like the timing service or the scheduling service, as a component in the system software layer in Figure 2 to design experiments for measurements. This view would be more appropriate when a real-time micro-kernel like EMERALDS [16] is used, where we retain only required services while switching off the others. Two basic RTOS services, timing and scheduling services, are measured. Timing services include various clock and timer management mechanisms implemented in an RTOS. The performance information for such a service is critically important to time-based activities in ESW. Scheduling services, on the other hand, are essential for software execution. The performance of such a service plays a key role in assessing the quality of the entire system. Timing and scheduling services are the basic services required by all embedded control applications and are supported in all RTOSs. Our measurements of timing and scheduling services should, therefore, reveal how these overheads and unpredictability change with different system configurations and application usages. When performance analysis needs to be done, such measured information can then be used to construct an accurate performance model based on the given application usage and system configuration.

To achieve such a measurement goal, we define a set of performance metrics for overhead and unpredictability

which are reusable for the analysis of a family of applications. We then construct synthetic workloads to enumerate representative ways of using timing and scheduling services in ESW applications and measure the service performance using microbenchmarks. The measurements were made on a set of selected hardware and RTOSs to demonstrate the general applicability of our measurement methods. Note that the performance of different hardware and RTOSs is not measured to compare and select targets. Instead, they are measured for constructing performance analysis models for a family of applications that will be executed on these targets.

4.1 Measurement strategy

Measurement environment. We designed a set of experiments for each different combination of hardware and operating system to obtain the performance of RTOS services directly. Table 1 lists the hardware we used in the measurements.

The RTOSs we measured include QNX 4.24, OSEKWorks 2.0 and RTLinux 3.0. We assume that the source code of these kernels is not available for the measurement, although the source code availability of RTLinux 3.0 helps us understand the relationships among the OS services, and can be used to verify our analysis results. The collected data were temporarily stored in the main memory and later dumped to an appropriate host for analysis after the services used for storage were turned on at the end of each measurement.

The versions of selected RTOSs that we had, could not be run on all the hardware platforms. The configurations used for measurements are given in Table 2.

Hardware	QNX	RTLinux	OSEKWorks
P133	QNX-P133	RTL-P133	-
P166	QNX-P166	RTL-P166	-
MPC555	-	-	OSEK-MPC

Table 2. Testbed configuration for measurements.

Performance metrics. It is important to choose the performance metrics for a given OS service carefully. The metrics are expected to be minimum in number, reusable for a family of applications, and independently measurable. A minimum set of metrics is desired to reduce the cost of experimentation and data collection while still providing sufficient data for the analysis. Finding a minimum set of performance metrics for a service requires understanding of the analysis requirements, dependencies among the OS services, and the relationships between different metrics. Reusability means that the metrics are measured once and reused whenever the same environment is used, thus eliminating duplicate measurements for the same environment for different applications with similar workloads and interaction patterns. Finally, the independently-measurable metrics simplify the experiments and data analysis and are more flexible when they are used in a performance model.

For the timing service, we measured clock overhead and interval jitter. The clock overhead is the CPU time used to process each signal generated from the system clock. The interval jitter is the variance in the length of time intervals. Interval jitter affects when periodic operations actually occur and is thus a source of unpredictability in the system. For the scheduling sevice, we measured the context switch overhead. The context switch overhead is defined loosely as in [3], which is the time taken from terminating a task to starting execution of another ready task.

Measurement tools and analysis. We use sampling tools to measure the specified performance metrics. To obtain accurate timing values, our measurement tool samples the processor clock cycles for each measurement. Most modern processors are equipped with some registers dedicated to performance and timing measurements. We use the hardware Time-Stamp Register (TSR) on the Pentium processor and the Time-Base Register (TBR) on the MPC 555. Both are 64-bit registers, initialized to 0 when the system powers up and incremented by 1 upon every hardware clock tick at the CPU speed.

For each experiment, 10,000 samples are collected during the normal execution. In addition to computing the average and standard deviation of the measured parameters, we also find the maximum and minimum values as performance bounds for each measured parameter on a given platform.

4.2 Experiment design

Experiments for timing service measurement: the metrics of timing services include clock overhead and interval jitter. The clock overhead depends mainly on the clock resolution. It can be measured by executing a test program under different resolutions. Specifically, given a program P with execution time e, the overhead of each clock tick can be computed using Eq. (1).

$$e_m = e + I_0 \cdot o + I_1 \cdot o + I_2 \cdot o + \dots \quad (1)$$

where e_m is the measured execution time of P; e is the real execution time of P; and o is the overhead of processing each clock tick. Each term $I_i \cdot o$ represents the overhead of processing the clock ticks during the time interval $I_{i-1} \cdot o$. I_i is the coefficient of the i-th order overhead. Given the

0-7695-1739-0/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

Hardware	Processor type	Processor speed (MHz)	Memory size (MB)	Cache size (KB)	Bus speed (MHz)
Intel P133	Pentium	133	32	128	66
Intel P166	Pentium	166	32	128	66
ETAS	MPC 555	40	2	-	20

Table 1. Hardware configurations for OS service measurements.

clock resolution r when the measurement is taken, I_i can be calculated recursively as:

$$I_0 = \lfloor \frac{e}{r} \rfloor, \quad I_1 = \lfloor \frac{I_0 \cdot o}{r} \rfloor, \quad I_2 = \lfloor \frac{I_1 \cdot o}{r} \rfloor, \quad \ldots \quad I_n = \lfloor \frac{I_{n-1} \cdot o}{r} \rfloor, \ldots \tag{2}$$

It can be seen from Eq. (2), I_i decreases exponentially. Thus, given any e, there exists a positive integer n such that for any $N > n$, $I_N = 0$, as the overhead introduced by its previous term will eventually be less then r. In the OS service measurements, the order of I_i seldom exceeds 2 as e is normally tens of milliseconds and the OS overheads are in the order of microseconds. Therefore, the clock overhead can be computed as follows:

$$e_m = e + \lfloor \frac{e}{r} \rfloor \cdot o + \frac{\lfloor \frac{e}{r} \rfloor}{r} \cdot o^2 \tag{3}$$

$$e_m = e + \lfloor \frac{e}{r} \rfloor \cdot o, \tag{4}$$

Eq. (3) is used when the clock resolution is fine (normally less than $0.5ms$) and/or the execution time is long, while Eq. (4) is used for the other cases. According to Eq. (3) and (4), the clock overhead can be finally derived using following equation:

$$o = \begin{cases} \frac{\sqrt{4 \cdot r \cdot \lfloor e/r \rfloor \cdot (e_m - e) + r^2 \cdot \lfloor e/r \rfloor} - r \cdot \lfloor e/r \rfloor}{2 \cdot \lfloor e/r \rfloor} & \text{for fine resolution} \\ \frac{e_m - e}{\lfloor e/r \rfloor} & \text{otherwise} \end{cases} \tag{5}$$

The execution time e of P is necessary to compute clock overhead o. We set the clock resolution much larger than e to obtain a measurement e_0 that is close to the real e, as given in Eq. (6).

$$e_0 = e, \qquad for \quad r \gg e \tag{6}$$

In our experiment, the test program P is designed with an execution time of $10ms$. The resolutions used for the clock overhead measurements range from $100\mu s$ to $100ms$.[1]

Interval jitter is a product of both the clock resolution and the application configuration. The timer resolutions used in our experiments are selected to be 500 μs and 1 ms.[2] The workloads in our experiment are designed to produce different patterns of timer intervals and different numbers of timers. Table 3 lists the values used in our jitter measurements.

Factor	values
clock resolution	500 μs, 1 ms
timer patterns	harmonic, non-harmonic
task priority	highest, medium, lowest
number of timers	1, 2, 5, 10, 15, 20

Table 3. Factors and values for interval jitter measurements.

A set of test programs are designed to perform the jitter measurements with the listed system attributes. Since only one timer can be associated with a process/thread in all the RTOSs studied, we need up to 20 tasks in the experiments.

Experiments for measuring the performance of scheduling service: the metric for measuring the performance of scheduling service is context switch overhead. The context switch overhead depends on the scheduling algorithm, the number of tasks in the ready queue, and the organization of the ready queue (e.g., sorted or unsorted). The priority-based preemptive scheduling algorithm is the one supported by all current RTOSs and used most frequently in ESW. So, our context switch overhead measurements are based on this scheduling algorithm. The task set ranges from 2 to 20 tasks. The measurements are taken between two specially-designed tasks in the task set. All other tasks, called *interference tasks*, are introduced only to change the length of the ready queue to learn the effect of the queue length on the context switch overhead. The task set is checked manually to be schedulable before taking the measurements. Table 4 shows these tasks and their attributes used for measuring the scheduling service performance.

task id	priority	period (ms)
1	2	1
2	1	triggered by task 1
3-20	3	1

Table 4. Attributes of experiment tasks.

[1]The resolution range is chosen based on the capacity test of a platform and the usage in applications.

[2]These values are chosen to reflect the fact that the clock overhead introduced by these values should be small but potentially has significant impact on jitter.

IEEE COMPUTER SOCIETY

The measurement with the given task set is designed as follows: Task 1 runs periodically with 1 *ms* period, and will trigger Task 2 upon its completion. The priority of Task 1 is lower than that of Task 2, but higher than all interference tasks. The TSR and TBR values are logged at both the end of Task 1 and the beginning of Task 2. Interference tasks run with the same period as Task 1. Thus, every 1 *ms*, all tasks but Task 2 are ready and will be moved to the ready queue. Since Task 1 has the highest priority in the ready queue, it executes first. After its completion, Task 2 is triggered and will be in the ready queue. Similarly, Task 2 becomes the highest priority task and will execute before any other task. Thus, Tasks 3–20 only affect the ready queue length during the measurement, and do not contribute any overhead to the context switch time between Task 1 and Task 2. The context switch overhead can then be obtained from the difference between the pairs of sampled values of Task 1's completion and Task 2's beginning. All interference tasks are assigned the same priority since their priorities have no effect on the measurements. Table 5 shows the number of tasks used for each measurement to learn the effect of the ready queue length.

test case	1	2	3	4	5
# of tasks	2	5	10	15	20
task in set	{1, 2}	{1, 2, 3-5}	{1, 2, 3-10}	{1, 2, 3-15}	{1, 2, 3-20}

Table 5. Task combinations for test cases.

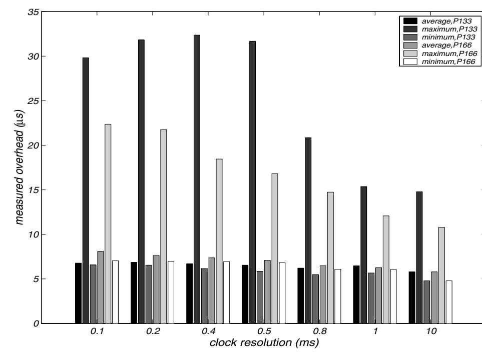

Figure 3. Timing service overhead of QNX.

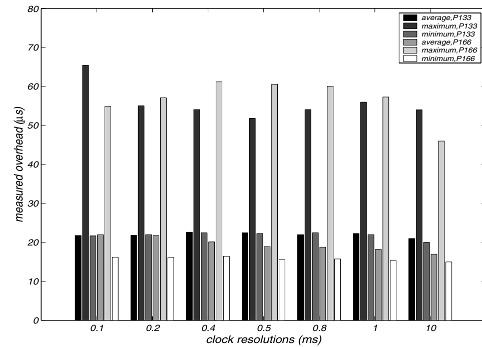

Figure 4. Timing service overhead of Real-Time Linux.

5 Measurement Results

5.1 Results of clock overhead measurements

The clock overhead at each resolution was computed using Eqs. (5). In the clock overhead calculation, we used the minimum execution time when the resolution is set to 100 *ms* as e_0 for each case. The computed clock overhead for all test cases are shown in Figures 3, 4 and 5.

The measurement results have shown that the clock overhead tends to decrease in general as the duration between clock ticks increases. This indicates that a fine-resolution clock will consume more system resources and may cause a schedulability problem, although such a clock may make the system more responsive. The quantitative effects of clock resolutions are OS-dependent. For QNX, the normal overhead is around $5 \sim 7\mu s$, but the maximum can be around $30\mu s$. The RTLinux overhead is around $20\mu s$ with the maximum at around $60\mu s$. The overhead for OSEKWorks is around $60\mu s$ except a higher overhead of $70\mu s$ is experienced when the clock resolution is set to 0.1 *ms*. We also experienced a system hang when the clock resolution was set to smaller than $50\mu s$ for QNX-P133, QNX-P166, $70\mu s$ for RTLinux-P133 and RTLinux-P166, and $80\mu s$ for OSEKWorks-MPC. The measured overhead for QNX and RTLinux is much less than the values required to make the system halt, while for OSEKWorks is very close. This indicates that the minimum available clock resolution depends on both OS implementation and hardware configuration.

Comparing the overheads of QNX and RTLinux on both platforms with those of OSEKWorks-MPC, one can see that the clock overhead of OSEKWorks was almost constant for any given resolution, while the maximum overheads for both QNX and RTLinux were significantly larger than the average for any given resolution. The less variant overhead for OSEKWorks may be due to the simple functionality of OSEKWorks and the flat memory structure of MPC555. Both help reduce unpredictability during execution.

5.2 Results of interval jitter measurements

We then measured the effects of clock resolution, the number and pattern of timer intervals, and task priorities on interval jitter. First, we are interested in how different clock resolutions affect the jitter of different interval lengths. The

0-7695-1739-0/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY

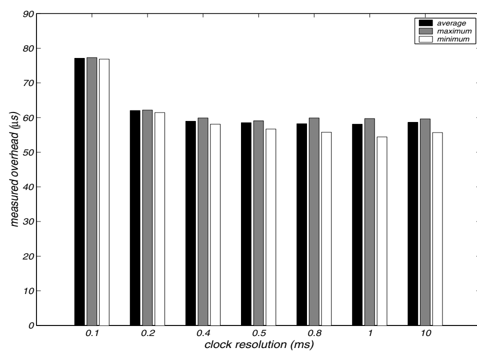

Figure 5. Timing service overhead of OSEKWorks.

measured results of interval jitter for QNX and RTLinux under different clock resolutions are plotted in Figures 6 and 7. For OSEKWorks, we did not observe any jitter for the examined clock resolutions.

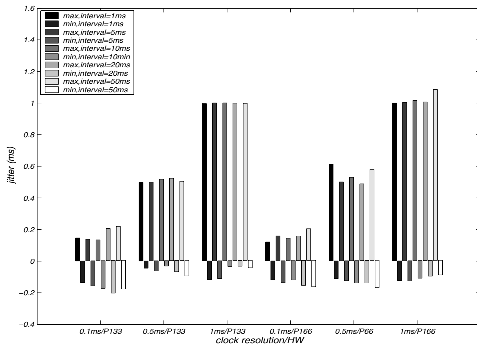

Figure 6. Interval jitter of QNX.

According to the measured results, the interval jitter varied greatly from one OS to another. In QNX, the interval jitter increased as the clock resolution became more coarse, while the jitter for RTLinux showed almost no change with different resolutions. The reason for this could be that QNX uses a clock-based scheduler while RTLinux and OSEKWorks use event-based schedulers. The lack of observed jitter for all experiments with OSEKWorks may also be the result of simple OS implementation and predictable hardware architecture.

The results also show that the interval jitter is independent of the interval length. This is different from the conventional understanding that the interval should be some relatively large multiples of the clock resolution to overcome the jitter. The clock resolution had a distinct effect on the magnitude of the interval jitter. As can be seen in Figure 6, the jitter was always bounded by twice the clock resolution. Figures 6 and 7 also indicate that using a faster

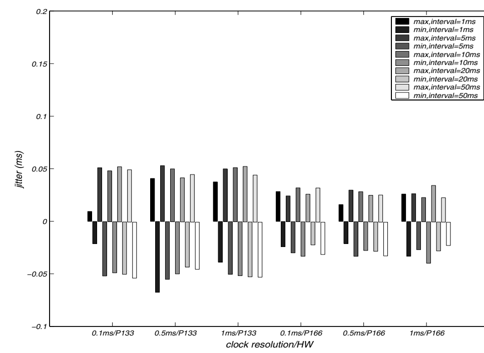

Figure 7. Interval jitter of RTLinux.

processor did not reduce the interval jitter for an OS using a clock-based scheduler, but reduced the jitter for the OS using an event-based scheduler.

Next, we studied the effects of the number of timers and interval patterns on jitter. The measurements also included the jitter experienced by tasks with different priorities. Figures 8 and 9 plot the measurement results where the intervals are harmonic and non-harmonic on QNX, respectively. Figures 10 and 11 show the results of the same experiments on RTLinux, while Figures 12 and 13 show the results of OSEKWorks.

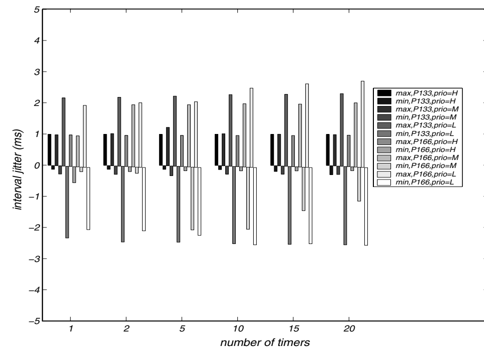

Figure 8. Interval jitter for harmonic intervals on QNX.

From these results, we first observed that the interval jitter increases with the number of timers in the system for all measured cases. Such dependencies should be an OS property and independent of hardware. Both cases of the same OS running on different hardware and different OSs running on the same hardware showed the same tendency of interval jitter changes. These results suggest that reducing the number of timers by combining tasks with the same intervals would reduce the interval jitter and consequently improve the system performance.

The interval jitter experienced by tasks with different pri-

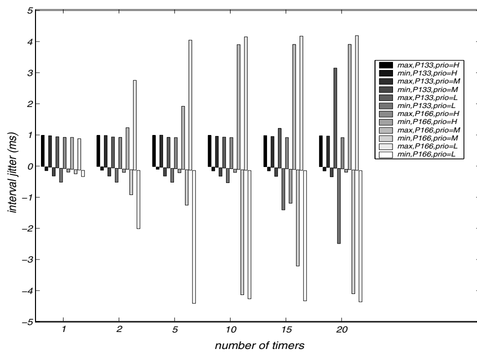

Figure 9. Interval jitter for non-harmonic intervals on QNX.

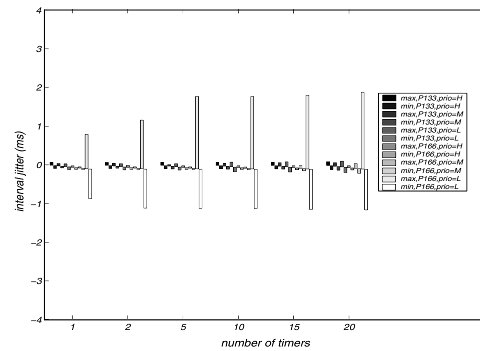

Figure 11. Interval jitter for non-harmonic intervals on RTLinux.

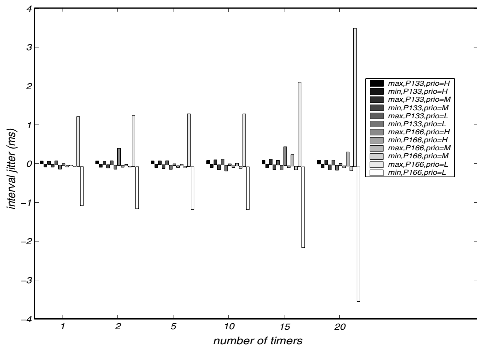

Figure 10. Interval jitter for harmonic intervals on RTLinux.

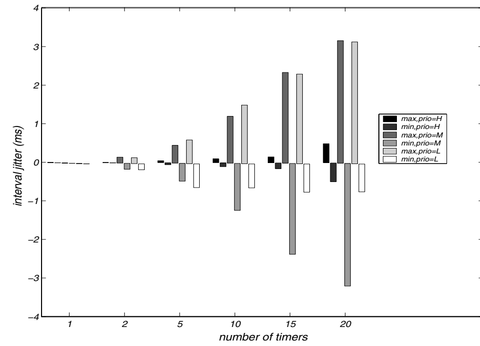

Figure 12. Interval jitter for harmonic intervals on OSEKWorks.

orities were also significantly different. A higher-priority task experienced a smaller jitter, while a lower-priority task experienced a larger jitter in all our measurements. Such observations are independent of the number of timers and interval patterns. The larger jitter experienced by a lower-priority task is likely the result of the cumulative effects of kernel activities that have a lesser effect on higher-priority tasks.

The interval patterns also had significant impact on interval jitter, and the impact depended on the of OS structure [7] and the number of timers in the system. Among the measured cases, jitter was almost the same for both harmonic and non-harmonic intervals when there were a relatively small number of timers (≤ 5). When the number of timers became larger, the jitter with non-harmonic intervals became larger for QNX, but showed the opposite for RTLinux and OSEKWorks. This implies that the clock-based OS implementation favors harmonic intervals, while the event-based OS implementation favors non-harmonic intervals.

5.3 Results of context switch measurement

The context switches were measured under the different configurations as described in Section 4. To learn the relationship between clock resolution and context switch time, we took measurements under different clock resolutions — specifically, 0.5 *ms* and 1 *ms* — as the OS scheduler can be either clock-based or event-based.

The measured context switch times of QNX, RTLinux, and OSEKWorks are shown in Figures 14, 15, and 16, respectively. For QNX, the average and minimum context switch times were not sensitive to the number of tasks in the ready queue, but the maximum context switch times increased when the number of tasks in the ready queue increased under a finer clock resolution. The difference between the system with 20 interference tasks and the system without any interference task can be as high as 300%. For RTLinux, both average and maximum context switch time increased as the number of tasks in the ready queue increased under any clock resolution, while the minimum times remain the same. Both average and maximum con-

0-7695-1739-0/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

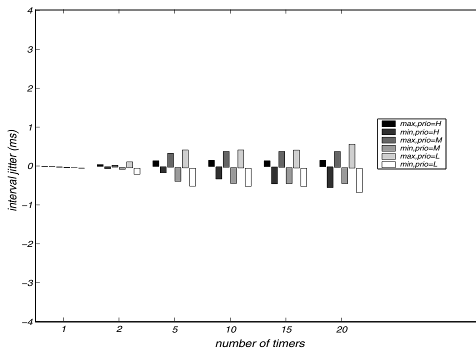

Figure 13. Interval jitter for non-harmonic intervals on OSEKWorks.

text switch times of the system with 20 interference tasks was twice as high as those for the system without any interference tasks. The context switch time for OSEKWorks showed little difference. These results imply that the context switch time depends heavily on the the RTOS implementation, and at least, will not increase if the number of tasks is reduced.

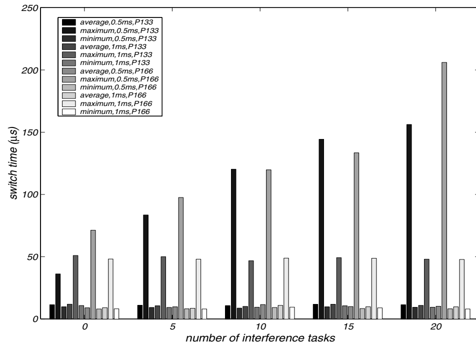

Figure 14. Measured context switch time for QNX.

We also observed that the clock resolution had a significant impact on context switch time. The context switch time (average, maximum and minimum) with a small clock resolution was higher than that with a larger resolution for QNX, while it was the opposite for RTLinux and OSEKWorks. This is because the context switch time of a clock-based scheduler may be more sensitive to the clock overhead, while the event-based scheduler may be more sensitive to the resolution.

6 Conclusions and Future Work

Most existing performance measurement methods consider either a fixed configuration with applications, or ignore how applications will use these services. Our proposed end-to-end measurement method is based on both microbenchmarks and synthetic workloads. The measured performance can therefore be stored with OS service components, and can be reused in performance analyses of different applications along with reuse of the OS services and their execution environments. We applied this method in measuring the timing and scheduling services of selected RTOSs, and presented several findings of the performance dependencies among the measured services, the service performance metrics and application configurations. Such information can be used in both timing analysis and schedulability analysis of integrated ESW.

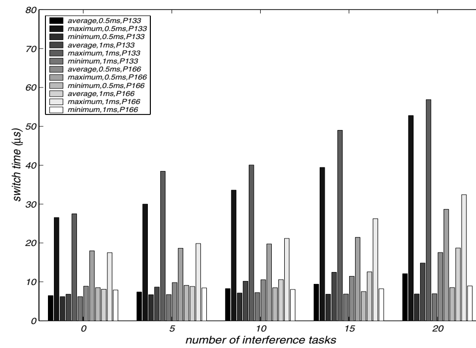

Figure 15. Measured context switch time for RTLinux.

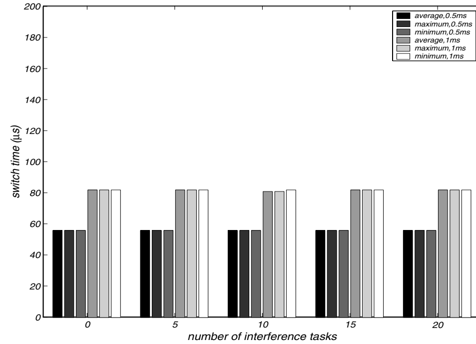

Figure 16. Measured context switch time for OSEKWorks.

Our future work will focus on using the measured information for performance analysis. We will apply the measured information to the analysis of both RTOSs and real applications, and validate the analysis results by comparing them with the simulation results. We will also integrate our work with other benchmarking methods for low-level OS services to construct the hierarchical performance model and build such measurement methods and results in

IEEE COMPUTER SOCIETY

ESW design toolkits to support performance-aware system design. The measurements of other system services such as communications and synchronization using the proposed method and their application to dynamic QoS management and online performance-aware reconfiguration will also be studied.

Acknowledgment

The authors would like to thank Advanced Dynamic International Company of Ann Arbor and PATH Project Group at UC Berkeley for their assistance in setting up our testbed.

References

[1] ARTiSAN Software Tools, Inc., I-Logix,Inc., Rational Software Corp., Telelogic AB, TimeSys Corp., and Tri-Pacific Software Inc. Response to the OMG RFP for schedulability, performance, and time. (revised submission). ftp://ftp.omg.org/pub/docs/ad/01-06-14.pdf, June 2001.

[2] K. Bradley. *A framework for incorporating real-time analysis into system design processes*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania, December 1998.

[3] A. B. Brown and M. I. Seltzer. Operating system benchmarking in the wake of lmbench: A case study of the performance of netbsd on the intel x86 architecture. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 214–224, June 1997.

[4] N. Frampton, J. Tsao, and J. Yen. Windows CE evaluation report: test plan and preliminary results. General Motors PowerTrain Group, April 1998.

[5] R. Jain. *The art of computer systems performance analysis: Techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons, 1991.

[6] R. Kar. Implementingthe rhealstone real-time benchmark. *Dr Dobb's Journal*, April 1990.

[7] K. A. Kettler, D. I. Katcher, and J. K. Strosnider. A modeling methodology for real-time/multimedia operating systems. In *Proceedings of the Real-Time Technology and Applications Symposium*, pages 15–26, May 1995.

[8] D. L. Kiskis and K. G. Shin. SWSL: A synthetic workload specification langauge for real time systems. *IEEE Transations on Software Engineering*, 20(10):798–811, October 1994.

[9] D. J. Lilja. *Measuring computer performance: A practitioner's guide*. Cambridge University Press, 2000.

[10] L. McVoy and C. Staelin. *lmbench*: Protable tools for performance analysis. In *Proceedings of the USENIX 1996 Annual TEchnical Conference*, San Diego, CA, January 1996.

[11] D. C. Schmidt, M. Deshpande, and C. O'Ryan. Operating system performance in support of real-time middleware. In *Proceedings of the 7th IEEE Workshop on object-oriented real-time dependable systems*, San Diego, CA, January 2002.

[12] B. Shirazi, L. Welch, B. Ravindran, C. Cavanaugh, B. Yanamula, R. Brucks, and E. nam Huh. DynBench: a dynamic benchmark suite for distributed real-time systems. In *Parallel and Distributed Processing: IPPS/SPDP Workshops*, pages 1335–1349, Berlin, Germany, April 1999.

[13] S. Toeppe and S. Ranville. RTOS evaluation and selection criteria for embedded automotive powertrain applications, 1999.

[14] S. Wang and K. G. Shin. An architecture for embedded softwre integration using reusable components. In *Proceedings of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, San Jose, CA, November 2000.

[15] N. Welderman. Hartstone: synthetic benchmark requirements for hard real-time applications. Technical Report CMU/SEI-89-TR-23, Software Engineering Institute, Carnegie Mellon University, 1989.

[16] K. Zuberi, P. Pillai, and K. G. Shin. EMERALDS: A small-memory real-time microkernel. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 277–291, december 1999.

IEEE COMPUTER SOCIETY