A Semi-Real-Time Scheduler for Service Robot Components on Windows NT Systems

Heejune Ahn*, Hyukjun Oh**, Yoojin Chung***

* Department of Control and Instrumentation Engineering, College of Engineering, SeoulTech (Seoul National University of Science and Technology), Gongneung-dong, Nowon-gu, Seoul 139-743, Republic of Korea
E-mail: heejune@seoultech.ac.kr

**Department of Electronics and Communication Engineering, College of Electronics & Information Engineering, Kwangwoon University, Wolgye-dong, Nowon-gu, Seoul 139-701, Republic of Korea
E-mail: hj_oh@kw.ac.kr (the corresponding author)

***Department of Computer Engineering, College of Engineering, Hankuk University of Foreign Studies, Yongin-si, Gyeonggi-do 449-791, Republic of Korea
E-mail: chungyj@hufs.ac.kr

Abstract

Recently, interest in service robots has been increasing in academia and industry, but the diversity of robot systems and the breadth of expertise required for them hinder progress in robot development. To overcome these challenges, many robot software frameworks have been proposed to enable code reuse and division of development efforts. However, the existing robot frameworks assume general, non-real-time operating system environment and pay little attention to real-time support, which is one of the vital requirements for service robots since they run real-time as well as non-real-time applications. This paper proposes a user mode scheduler of periodic real-time components on unmodified Window NT operating systems on which most service robots are built. The proposed scheduler uses real-time characteristics of the general purpose Windows NT: OS timer resolution control, a high resolution multimedia timer, and minimal real-time priority support. The performance limits of the scheduler are examined by measuring the influence of the kernel mode operation. The DC motor PID control experiment result demonstrates that our proposed scheduler can achieve 10 ms period component executions with 20% accuracy.

Key Words Autonomous Robots, Real-Time Scheduling, Windows NT, Component Based Design

1. Introduction

Recently, interest in service robots has been increasing in academia and industry. A service robot can be defined as a robot providing various services for the well-being of humans, society, and equipment outside industrial automation applications [1, 2]. However, developing service robot software is challenging, because robots have widely varying hardware and require broad expertise, starting from control and driver level software and continuing up through perception, reasoning, and beyond.
To meet these challenges, many robot software frameworks have been proposed [3]. While each framework has particular purposes, one common emphasis is how to enable code reuse and division of development efforts. Due to its modularity, reusability, and productivity, the component-based approach has been adopted for robot software frameworks. For example, OMG's RTC (Robot Technology Component) [4], Microsoft's MRDS (Microsoft Robotics Developer Studio) [5], ROS (Robot Operating System) [6], and Korea OPRoS (Open Platform for Robotic Services) [7, 8, 9] are all based on the component-based approach.

On top of software reusability, real-time support is essential but challenging to service robots. In contrast to electronic information devices like handsets and set-top boxes, service robots operate in the human working space and interact mechanically with humans and the surrounding environment. However, to our knowledge, the existing frameworks pay little or very limited attention to real-time support. In this paper, we propose a user mode real-time component scheduler for an OPRoS framework.

However, due to the similarities in program control management, the scheduling algorithm can be easily integrated with most component-based robot frameworks on Windows NT based frameworks.

Since the operating system plays the key role in supporting real-time service, an RTOS (Real-Time Operating System) ought to be the natural choice for service robot operating systems [10]. However, many service robots have been developed on Windows NT-based operating systems, such as Windows 2000, Windows XP, Windows Vista, and Windows 7. This is mainly because Windows NT has a large developer community, hardware and device driver support, rich available application libraries, and development tools. For these reasons, Windows NT’s real-time service capability has been studied extensively [14, 15, 16, 17, 18]. Three architectural approaches have been considered for real-time support of service robots based on a Windows NT system.

First, non-real-time applications are located on a Windows NT main board system, whereas real time applications are located on a firmware-based or RTOS-based subsystem. For example, long-term navigation is done on Pentium Windows NT, and motion control of wheels is on a daughter DSP board.

Second, the kernel is modified for real-time support. Since we cannot change the kernel of Windows, a device driver extension technique is often performed, such as in RTX [18]. The RTX approach is very similar to the techniques in RTLinux and MontaVista Linux. In spite of better real-time service, the kernel approach requires great
care in developing application code and deprives the programmers of the rich user level libraries of operating environments.

In the third, which is our approach, Windows NT is used as it is. On top of a few known techniques for providing a high resolution timer in the application process [15, 16, 17], a component scheduler in user mode delicately controls the execution threads and API usages, taking the component's timing requirement into account. We investigate the practical conditions and range in which an application-level approach can achieve the real-time requirements of service robot systems. Although real-time support using Windows NT is still controversial, the analysis and experiments in this paper show that the proposed scheduling mechanism can achieve 10 ms period component executions with 20% timing accuracy.

The remainder of the paper is organized as follows. Section 2 provides an overall description of the OPRoS component execution model and its component scheduling requirements. Section 3 presents the user mode execution scheduler architecture for real-time component scheduling. Section 4 provides verification experiments and performance results. Section 5 concludes this paper with a few future work directions.

2. Robot Software Framework
2.1 OPRoS Component Model

A detailed description of the OPRoS framework specifications may be required for an implementation-level understanding of our proposed mechanisms, but the readers can understand the overall operation using Figure 1 and the following quick summary. For a detailed description, refer to the specifications [9], overview paper [8], and the project website [7], where you can download the implementation source. The framework, normally a process in an operating system, contains multiple application components. Typical service robots include a sensor, an actuator, and various algorithm components.

The framework provides services such as thread, lifecycle management, configuration, and inter component communication. The framework manages the lifecycle of the components, such as loading/unloading the component library and creating/destroying an instance, the state of the component using lifecycle interface functions, ‘initialize(),’ ‘start(),’ ‘stop(),’ ‘destroy(),’ ‘recover(),’ ‘update(),’ and ‘reset(),’ and executes jobs by invoking the callback functions defined in the user components, that is, ‘onExecute()’ and ‘onEvent().’ A component can communicate only through ports, which are classified into data, event, and service port types according to the synchronization and argument styles.
When its callback functions are called, a component can execute its jobs and communicate with other components only through ports. The component configuration information is provided in an XML file called a ‘component profile.’ In addition, the components are grouped into application tasks in another XML file called an ‘application profile.’ The connectors and adaptors for communication middleware have little to do with the subject of this paper, so we give no further description here. Furthermore, the OPROS framework shares an architecture similar to other robot software frameworks. Thus, most of the findings in this paper can also be applied to them.

![Fig. 1. OPRoS component model and framework architecture [7].](image)

### 2.2 OPRoS Execution Model and Scheduler Requirements

An OPRoS component can be executed in either a periodic or non-periodic execution style. In this paper, we only consider scheduling for periodic real-time components. This is because real control systems are modeled by a periodic digital control loop of sensor, algorithm, and actuator [10, 11, 12, 13]. For example, the DC motor control of a mobile robot wheel consists of an encoder sensor, a PID control logic, and a PWM actuator component.

The OPRoS framework schedules the execution timing of components by periodically calling the callback function, ‘onExecute().’ Here two scheduling constraints should be satisfied. First, the invocation should be punctual lest the delay execution destroy the control stability or the response is delayed excessively. In real-time system theory, the task is defined by three parameters, (period, execution time, deadline) [10]. Often in a control system, the deadline time of a task is the same as its period, which is also the case considered in this paper.

Furthermore, the components from the same application have input-output dependency, the so-called ‘precedence condition.’ In the above control example, the encoder sensor should be executed before the PID algorithm component, which should also be executed
in advance of the PWM actuator component.

3. Semi-Real-Time Component Scheduler Design

3.1 Windows NT’s Real-Time Characteristics

Since Windows NT was first announced in 1993, it has been the kernel for the most widely used personal computer operating systems, Windows 2000, Windows XP, Windows Vista, and Windows 7. Certainly, Windows NT is a general operating system kernel that focuses on overall throughput optimization and non-real time applications. However, somewhat surprisingly, Windows NT has many RTOS features [15, 20], which are summarized below.

According to Timmerman [15], a true RTOS must satisfy the following four requirements: 1) preemptive multi-threading scheduling, 2) deterministic priority, 3) deterministic thread synchronization, and 4) priority inheritance for prevention of priority inversion. As for the first and second conditions, Windows NT provides 16 static preemptive priorities (16 to 31) as well as 16 dynamic non-preemptive priorities (0 to 15). As for the third condition, various thread synchronization mechanisms, such as a critical section, an event, a semaphore, and a mutex, are well supported. Last, Windows NT does not support priority inheritance, but priority inheritance is often not supported by simpler commercial RTOSs, such as uCOS-ii [21]. In addition, virtual pages can be selectively locked up for not paging-out the real-time code and data sections.

On the other hand, aside from the lack in the priority inheritance mentioned above, some features are against real-time support. The Windows NT kernel does not provide kernel mode preemption. The priority has an increasing order of ISR, DPC, kernel and real-time priority threads, non real-time priority threads. Hardware interrupts are served in two phases for higher responsiveness: first, a normal ISR (interrupt service routine) quickly handles an interrupt and the remaining processing of the interrupt is done in DPC (Deferred Procedure Call). Because DPCs have higher priority than real-time threads, the execution time is affected by IO traffic, such as hard disk access time, USB packets, and network traffic. The number 7 of real-time priority is too limited, i.e., priority values (16, 22, 23, 24, 25, 26, 31).

With these understandings in mind, in Figure 2 we summarize the timing diagram that illustrates three delay sources, timer accuracy latency, DPC latency, and task scheduling latency. Windows NT uses HAL (Hardware Abstraction Layer) for OS timer interrupts. The normal interrupt time, the so-called OS tick is 15 ms but can be reduced to 1 ms for
a support multimedia application like MIDI. The timing accuracy of task synchronization can be limited to this OS tick value, that is, up to +/-1 ms jitter. Furthermore, from a previous study [15] and our measurements, one more artifact is noted. Though the MSDN denotes the time units in milliseconds, the real value is just approximated by the OS tick. When a 1 ms timer interrupt is setup in the HAL (Hardware Abstraction Layer), the exact duration is 976 us. This is because the interrupt is divided by $2^{10}$, i.e., 1024. This is illustrated in Figure 3 and the measurement result is reported in Section 4.

Fig. 2. Timing diagram for Windows NT task scheduling.

Fig. 3. The origin of periodic jitter in Windows multimedia timer intervals.

### 3.2 Windows NT's Real-Time Characteristics

Before describing our scheduling algorithm, we examine the current scheduler implementation [9] for comparison. In the component specification, how to map the component to threads is not defined. In the reference implementation, all the components with the same execution period are assigned to the same executor, a Windows NT thread. We assume in the following examples that executor 1 has a shorter period and deadline than executor 2. For example, the components for executor 1 are mechanical control components and the ones for others executors are human interface or an intelligent perception algorithm.

Figure 4 illustrates the timing diagram that emphasizes the scheduling issue. The executor records times before and after executing all the components, i.e., calls the ‘onExecute()’ callback methods, and calculates the time sleep time...
Since the number of execution periods can be more than several tens, which is much larger than the Windows NT real-time priority number, which is only 7, each executor cannot have a separate priority. Therefore, preemption is not possible. Executor 1 with a shorter period is delayed by the execution of executor 2 with a longer period. Furthermore, the windows API `Sleep(long ms)' does not provide enough accuracy in general. In addition, a multi-core CPU and the precedence constraint are not considered in the algorithm.

Fig. 4. Timing diagram for component execution in OPRoS Reference Implementation [9].

3.3 Proposed Scheduling Algorithm

To meet the real-time timing requirements for executors under the constraints of Windows NT, we propose a two-level scheduler architecture: the ‘inter-executor’ and ‘inner-executor’ scheduler. The inter-executor scheduler suspends and resumes executors based on the status of the executors. Note that the scheduler threads can preempt and control execution due to the preemptive scheduling of Windows NT for real-time priority threads. For this purpose, the inter-executor scheduler is a user thread with the highest priority of 31, i.e., ‘THREAD_PRIORITY_TIME_CRITICAL’ of ‘REALTIME_PRIORITY_CLASS.’

Figure 5 illustrates a time sequence for inter-executor scheduling. The scheduler sleeps while waiting for ‘scheduling events’ from a Windows multimedia timer and executor threads. In the multimedia timer, the highest priority is also setup and wakes up at every 1 ms interval, which is officially the smallest value in Windows NT. Since the scheduling time accuracy is obtained from the timer interrupt accuracy, the OS tick resolution is set to the minimum value in the current Windows kernel, i.e., 1 ms, using ‘timeBeginPeriod()’ API. The ‘timeBeginPeriod()’ call reconfigures the kernel's OS tick time and the corresponding scheduling interval. Therefore, the disadvantage of a 1 ms
time resolution is the OS context switch and scheduling overhead. However, we observed that most recent computer hardware is more than adequate for this level scheduling interval.

Whenever the multimedia timer thread wakes up, it signals a scheduler event. In addition, an executor thread calls ‘yield()’ when it finishes calling all the dedicated components. The ‘yield()’ call also signals the scheduler thread. The ‘yield()’ call should be implemented with a great care, since the call should make the calling thread/executor block itself, and the scheduler thread should get the cpu cycle in any case. The simple ‘Sleep()’ call in Windows does not guarantee this, because several threads can run simultaneously in a multi-core system. Furthermore, even in a single core system, the ‘Sleep()’ call does not guarantee the real yield operation. In order to solve this artifact, the scheduler thread calls ‘WaitForMultipleObjects()’ for all events set by the yielding threads for each CPU core and the multimedia timer events. In our implementation, we add a ‘busy-while-sleep’ loop after yielding its thread to the scheduler to ensure the yield operation.

Fig. 4. Proposed inter-executor scheduler operation.

3.4 Precedence Constraint

If components with a precedence relationship are assigned to different CPU cores, the executor thread needs to check whether or not the precedent component is executed before its component, that is, thread synchronization [19]. This constraint would result in a complicated scheduler and low system efficiency. In the proposed scheduler, therefore, the execution order is guaranteed by assigning components from the same application to the same CPU core. The executor sorts the components in the combined order of application ID and internal execution order. This sorting mechanism provides
one more benefit: each component has relatively less dynamic execution time from the
start of the executor start time.

When the component execution periods of an application are different, the order needs
to be defined [23]. We assume that the longer execution period component is executed
after the smaller. Table 1 illustrates the execution order of components by our scheduler
with a sensor and actuator components with a 10 ms execution period and a control
algorithm component with a 20 ms execution period.

Table 1. Execution order of components with different execution periods

<table>
<thead>
<tr>
<th>Round</th>
<th>Sensor (10 ms)</th>
<th>PID (20 ms)</th>
<th>Actuator (10 ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round 1</td>
<td>1st</td>
<td></td>
<td>2nd</td>
</tr>
<tr>
<td>Round 2</td>
<td>3rd</td>
<td>5th</td>
<td>4th</td>
</tr>
<tr>
<td>Round 3</td>
<td>6th</td>
<td></td>
<td>7th</td>
</tr>
<tr>
<td>Round 4</td>
<td>8th</td>
<td>10th</td>
<td>9th</td>
</tr>
</tbody>
</table>

3.5 Admission Control Rule

The deterministic operations of an RTOS enable us to determine the real-time
execution feasibility of the system. As our scheduler is based on an RMS (Rate
Monotonic Scheduler), the RMS feasibility inequality [19] can be applied. Whenever a
new component with period and execution time is added to the system, the RMS checks
the feasibility for each CPU core after assigning it to the least loaded CPU core.

\[ U_{\text{CORE}} + \frac{C}{T} = \sum_{i=1}^{n_{\text{CORE}}} \frac{C_{\text{CORE}(i)}}{T_{\text{CORE}(i)}} + \frac{C}{T} < n_{\text{CORE}}(2^{n_{\text{CORE}}-1}) \text{ for } CORE = 1, \ldots, n_{\text{CORE}}. \] (2)

Note that equation (2) is the worst case estimation for all possible periods and
execution times. The main focus of this paper is the embedded scheduler design in a
Windows NT kernel and RMS is known as mostly used in industry for its simplicity and
low computation. Other scheduling policies, such as EDF (Early Deadline First) or
Cycle-based Scheduling, can be applied for the admission control and the scheduling
itself.

4. Experiment and Performance Evaluation

We performed experiments on the OPRoS RI, with modification of the proposed
scheduling method, on Windows XP SP3 and an Intel Pentium Dual core CPU 2.66 GHz.
The timing measurement was obtained by Windows API, ‘QueryPerformanceCounter()’
and ‘QueryPerformanceFrequency(),’ which are known to provide the most accurate
timing information in a Win32 environment. The data was recorded onto memory with the measurement artifacts being minimized, and afterward saved to a file and processed off-line. The following sampled results are for functional verification and for determining the numerical limits of the proposed system.

4.1 Timer Accuracy

Figure 6 compares the accuracy of the multimedia timer in the proposed scheduler with the sleep mechanism in the reference implementation. As expected, the sleep mechanism suffered from the inaccuracy of ‘Sleep()’ API. The multimedia timer provided much better accuracy, but non-uniformity of the multimedia timer intervals due to the inaccuracy of the OS timer was observed. The origin of this inaccuracy is in setting the minimum OS tick value to 1 ms; 976 us is the real time period as mentioned in Section 3.1. Therefore, the first 41 multimedia timer callbacks arrive at intervals of 0.976 ms, and the 42nd occurs at 0.976 plus 0.976, i.e., 1.952 ms. The results reveal that up to 1 ms deviation is inevitable in the vanilla Windows NT system. Since many practical control systems allow 10 to 20 percent accuracy for the timer, we used the minimum 10 ms-period in the following experiment. Other researchers may be interested in trying to solve this artifact later.

![Fig. 6. Event start time of execution interval comparison with external disturbance.](image)
(Left: the proposed scheduler case, right: the reference implementation case)

4.2 Effects of External Loads on Scheduling Precision

The experiment in Figure 6 is the performance when only one component is running without any disturbance from the application and kernel. The proposed algorithm is designed to control the user mode application, but not kernel mode activities. To examine the performance of the scheduler under such disturbances, intensive experiments were performed to estimate the effects of disturbance due to the DPCs, since we could not calculate the effects of a DPC analytically. We generated a full-load
network (approximately 32 Mbps), USB traffic (approximately 100 Mbps), and file IO on three different configurations: (a) no external load, (b) full external load and normal priority thread, and (c) full external load and real-time thread.

Figure 7 shows the time variation of the task scheduling compared with the case of the reference implementation. With a timing tolerance of 10%, more than 98% of the scheduling time satisfied the deadline for the no-load, full-load, and real-time thread cases. However, in the full-load and normal thread priority cases, many more scheduling misses were observed.

In the previous study [15], it was also reported that an ill-designed device driver can make things worse, so practically one should check if the robot system has such ill-performing drivers. Fortunately most commercial device drivers seem to perform well, but the Windows Certificate Log does not guarantee it and the Windows kernel has no mechanism for preventing a device driver from running a long time.

![Fig. 7. Event time of execution interval (top: the proposed scheduler case, bottom: the reference implementation case)](image)

4.3 Real Application: DC Motor Control

In order to demonstrate the scheduler performance, a typical PID DC motor speed control application with an encoder sensor, PID control algorithm, and PWM output actuator was performed (Figure 8). The system parameters for the motor control system are listed in Table 2. The PID gain parameters were tuned by the well-known Ziegler and Nichols's algorithm [22]. The control result by an interrupt-based programming AVR micro-controller was used for the optimal reference. Two applications, DC motor control and vision processing, were implemented on the original and modified OPRoS engine. The DC motor control consists of a UART component for interfacing with the AVR motor driver, a PID control component, and reference position generation components, which repeats a 180° rotation command to the PID components. The vision component was chosen because it is an essential and computation-demanding
application for service robots. The vision component got a webcam image through a USB interface, and performed typical image processing in a 100 ms period.

In Figure 9 and Table 2, the motor speed control by OPRoS components under the proposed scheduler is identical to the one by the dedicated micro-controller based firmware. This result shows the proposed scheduler works successfully for this range period control task. We compared the encoder outputs with the proposed scheduler and reference implementation. There was no difference when the input and system bandwidth was larger than about 100 ms according to the famous Nyquist sampling theorem. When the input period was less than a critical value, the reference implementation scheduler lost stability and started oscillating, while the proposed scheduler showed stable control.

![Figure 8. Experiment setup for DC motor control](image)

**Table 2. DC motor & PID control parameter for the proposed scheduler performance test**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPM</td>
<td>172 rpm</td>
</tr>
<tr>
<td>Encoder</td>
<td>3584 pulse/round</td>
</tr>
<tr>
<td>Timing</td>
<td>10 ms</td>
</tr>
<tr>
<td>Kp</td>
<td>6</td>
</tr>
<tr>
<td>Ki</td>
<td>40</td>
</tr>
<tr>
<td>Kd</td>
<td>0</td>
</tr>
</tbody>
</table>

![Fig. 9. DC motor PID control test for the proposed scheduler on OPRoS system.](image)
5. Conclusion

The existing robot software frameworks do not provide real-time service. This paper proposed a user mode scheduler of periodic real-time components on unmodified Window NT operating systems on which most service robots are built. The proposed scheduler uses a few real-time characteristics of the general purpose Windows NT: OS timer resolution control, a high resolution multimedia timer, and minimal real-time priority support. We also investigated the effects on our scheduler of inaccuracies of OS timer interrupts and the kernel level interrupt handling mechanism (DPC). The analysis and experiments in this paper will be pleasing to those use Windows NT in service robot development, because the proposed scheduling mechanism can achieve 10 ms period component executions with 20% timing accuracy, which is the commonly used control period in service robot systems. We demonstrated this using a real DC motor control example. Our team is developing the fault tolerant and fault safe software framework based on OPRoS for service robots. Our initial results can be found in [24]. We are integrating the prototype robot system with the real-time component scheduler developed in this paper.

6. Acknowledgments

This work was supported in part by the Korea MKE, under the Strategic Technology Development Program (No-10030826), the IT R&D program of MKE/KEIT (K1002145), and the Research Grant of Kwangwoon University in 2010.

References

[23] Ferrari, A., Natale, M. D., Gentile, G., Reggiani, G., and Gai, P., Time and memory tradeoffs in the implementation of AUTOSAR components, DATE 2009, Nice, France,

*Corresponding author: Hyukjun Oh, Ph.D.
Department of Electronics and Communication Engineering, College of Electronics and Information Engineering, Kwangwoon University,
Wolgye-dong, Nowon-gu, Seoul 139-701, Republic of Korea
E-mail: hj_oh@kw.ac.kr

Heejune Ahn received his PhD, MS, and BS degrees in electrical engineering from KAIST, Republic of Korea, in 1999, 1995 and 1993, respectively. He is now an associate professor of the Department of Control and Instrumentation Engineering at Seoul National University of Science and Technology (SeoulTech), Seoul, The Republic of Korea. He has been a GSM/GPRS/UMTS wireless mobile protocol software engineer at LG Electronics Inc., Korea, from 2000 to 2002. From 2002 to 2003 he worked as a software architect at Tmax Soft Inc. His research interests include multimedia communications, protocol development, network system performance analysis, and real-time and embedded systems.

Hyukjun Oh received his PhD, MS, and BS degrees in electrical engineering from KAIST (Korea Advanced Institute of Technology), Daejeon, The Republic of Korea, in 1999, 1995 and 1993, respectively. He is an associate professor of the Department of Electrical Communications Engineering at Kwangwoon University, Seoul, The Republic of Korea. From 2001 to 2004, he worked on 3GPP MSM development at Qualcomm Inc., USA. His areas of research include wireless communication system, OFDM and CDMA communications.

Yoojin Chung received the B.E., the M.S. and the Ph.D. degrees in Computer Science and Engineering from Seoul National University, Seoul, Korea, in 1989, 1991 and 1997, respectively. She has been with the Department of Computer Engineering, Hankuk University of Foreign Studies since 2001, where she is currently a professor. She was a department chair during 2005 and 2006 and a visiting professor at the University of California, Irvine from 2007 to 2008. Her research interests include embedded systems, computer security, algorithms, information retrieval, and bioinformatics.