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 meet these challenges, many robot software frameworks have been proposed. Although service robots have real-time as well as non-real-time applications, the existing robot frameworks pay little attention to real-time support. 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 utilizes real-time characteristics of the general purpose Windows NT: OS timer resolution control, 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.

Categories and Subject Descriptors
D.4.7 [Organization and Design]: Real-time systems and embedded systems

General Terms
Performance, Design, Reliability

Keywords
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]. 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 MSRDS (Microsoft Robotics Developer Studio) [5], ROS (Robot Operating system) [6], and Korea’s OPRoS [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 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 function, 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 [11, 12, 13, 14, 15]. 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 one cannot change the kernel of windows, a device driver extension technique is often performed, such as in RTX [15].

In the third and 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 [12, 13, 14], 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.

2. FRAMEWORK MODEL
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 the specifications [7], overview paper [8], and reference implementation [9], 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’ Also, 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.

The OPRoS framework schedules the execution timing of components by periodically calling the callback function, onExecute(). Here two scheduling constraints should be satisfied. Firstly, 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. PROPOSED SCHEDULER
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 [11], [17], which are to be summarized below.

According to Timmerman [10], 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 from 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 RTOs such as uCOS-II [18]. 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, 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 of real-time priority is quite limited to ‘7’ (16, 22, 23, 24, 25, 26, 31).

With these understandings in mind, in Figure 2 we summarized 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 [12] 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. The measurement result is reported in Section 4.

$$T_{SLEEP} = T_{PERIOD} - (t_{AFTER} - t_{BEFORE})$$  \hspace{1cm} (1)

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. In Figure 3, 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.

![Figure 3. Timing Diagram for Component Execution in OPRoS RI [8].](image)

### 3.3 Proposed Scheduling Algorithm

To meet the real-time timing requirements for the executor 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 or REALTIME_PRIORITY_CLASS.

Figure 4 illustrates a time sequence for inter-executor scheduling. The scheduler sleeps while waiting for ‘scheduling events’ from a 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 the smallest value in Windows NT. 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. Furthermore, 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.

![Figure 4. Inter-Executor Scheduler Operation.](image)
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 components are executed before their component, that is, thread synchronization [16]. 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 executor start time.

When the component execution periods of an application are different, the order needs to be defined [20]. We assume that the longer execution period component is executed after the smaller. Figure 5 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>
<thead>
<tr>
<th>Execution (10 ms unit)</th>
<th>Sensor (10m)</th>
<th>PID (20ms)</th>
<th>Actuator (10ms)</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>

Figure 5. Execution order of components with different execution periods

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 [16] 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 them to the least loaded CPU core.

\[ U_{\text{CORE}} + \frac{C}{T} = \sum_{i=0}^{n_{\text{CORE}}} \frac{C_{\text{CORE}}(i)}{T_{\text{CORE}}(i)} + \frac{C}{T} < n_{\text{CORE}} \left(2^{n_{\text{CORE}}} - 1\right) \]

for \(\text{CORE} = 1, \ldots, n_{\text{CORE}}\) (2).

Note that eqn. (2) is the worst case estimation for all possible periods and execution times.

4. EVALUATION OF THE EXPERIMENT

Experiments were performed on the OProS RI with modification of the proposed scheduling method. The experiments were performed on Windows XP SP3 and an Intel Pentium Dual core CPU 2.66 GHz. The timing measurement was obtained by Windows API, ‘QueryPerformanceCounter’ and ‘QueryPerformanceCounterFrequency’, which are known to provide the most accurate timing information in a Win32 environment. The data was recorded onto memory while minimizing the measurement artifacts, 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 examines the accuracy of the multimedia timer in the proposed scheduler and the sleep mechanism in reference implementation. As expected, the sleep mechanism suffered from the inaccuracy of ‘Sleep’ API. The multimedia timer provided much better accuracy, but the 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 say 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 applications thereafter.

Figure 6. Event Start time of Execution Interval Comparison with External Disturbance. (Left: the Proposed Scheduler Case, Right: the Reference Implementation Case)

4.2 Effects of External Loads on Scheduling Precision

The experiment in Figure 5 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 disturbance, intensive experiments were performed to estimate the effects of disturbance due to the DPCs, since we could not calculate the effects of DPC analytically. We generated a full-load network (~32 Mbps) and USB traffic (~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 and full-load and real-time thread cases. However, in the full-load and normal thread priority cases, much more scheduling miss is observed.

In the previous study [12], 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.

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 1. The PID gain parameters were tuned by the well-known Ziegler and Nichols’ algorithm [19]. The control results 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, 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 is no difference when the input and system bandwidth is larger than about 100 ms according to the famous Nyquist sampling theorem. When the input period becomes less than a critical value, the reference implementation scheduler loses the stability and start showing damping with oscillation, while the proposed scheduler shows stable control.

5. CONCLUSION

The existing robot software frameworks do not provide real-time function. 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 utilizes a few real-time characteristics of the general purpose Windows NT: OS timer resolution control, 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 are pleasing to who want 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.

![Figure 7. Event time of Execution Interval (Top: the proposed scheduler case, Bottom: the reference implementation case)](image7)

![Figure 8. Experiment Setup for DC motor Control](image8)
Table 1. 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>

Figure 9. DC motor PID control Test for the proposed scheduler on OPRoS system.

6. ACKNOWLEDGMENTS

This work was supported in part by the Korea MKE, under the Strategic Technology Development Program (No-10030826), by the IT Convergence Research Institute, Seoul National University of Technology, and by Seoul R&D Program (10560) and the IT R&D program of MKE/KEIT (K1002145)

7. REFERENCES