Future Generation Computer Systems 56 (2016) 171–178

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

RT-ROS: A real-time ROS architecture on multi-core processors

Hongxing Wei [a,1], Zhenzhou Shao [b], Zhen Huang [a], Renhai Chen [d], Yong Guan [b], Jindong Tan [c,1], Zili Shao [d,*,1]

[a] School of Mechanical Engineering and Automation, Beihang University, Beijing, 100191, PR China
[b] College of Information Engineering, Capital Normal University, Beijing, 100048, PR China
[c] Department of Mechanical, Aerospace, and Biomedical Engineering, The University of Tennessee, Knoxville, TN, 37996-2110, USA
[d] Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China

ARTICLE INFO

Article history:
Received 6 February 2015
Received in revised form
20 April 2015
Accepted 12 May 2015
Available online 9 June 2015

Keywords:
Real-time operating systems
Robot Operating Systems
Multi-core processors

ABSTRACT

ROS, an open-source robot operating system, is widely used and rapidly developed in the robotics community. However, running on Linux, ROS does not provide real-time guarantees, while real-time tasks are required in many robot applications such as robot motion control. This paper for the first time presents a real-time ROS architecture called RT-RTOS on multi-core processors. RT-ROS provides an integrated real-time/non-real-time task execution environment so real-time and non-real-time ROS nodes can be separately run on a real-time OS and Linux, respectively, with different processor cores. In such a way, real-time tasks can be supported by real-time ROS nodes on a real-time OS, while non-real-time ROS nodes on Linux can provide other functions of ROS. Furthermore, high performance is achieved by executing real-time ROS nodes and non-real-time ROS nodes on different processor cores. We have implemented RT-ROS on a dual-core processor and conducted various experiments with real robot applications. The experimental results show that RT-ROS can effectively provide real-time support for the ROS platform with high performance by exploring the multi-core architecture.

1. Introduction

ROS, an open-source robot operating system, has been being rapidly developed and widely used in the robotics community [1]. Based on the ROS framework, many researchers have developed their software for diverse robots such as Barrett WAM [2] and Raven-II [3]. However, ROS runs on Linux, and cannot provide real-time guarantees. This limits its usage, as real-time tasks are required in many robot applications such as robot motion control. Therefore, it becomes a key issue to make ROS be real-time. On the other hand, multi-core processors offer a promising platform for robot applications. Compared to the traditional robot computing platform with separated host and guest systems, a multi-core processor can provide more powerful computing capacity and less communication overhead. Thus, it is also vitally important to effectively run ROS on multi-core processors by exploring the multi-core architecture for robot applications. This paper focuses on solving the problem of making ROS be real-time with high performance on multi-core processors.

To make ROS be real-time, a common approach is to run real-time tasks on guest embedded systems and run non-real-time tasks on a host system such as in ROS Industrial and ROS *Bridge* [4]. However, by separating ROS tasks into different computing systems, it not only introduces big communication overhead but also increases the manufacturing cost. In fact, a multi-core processor such as Intel Pentium multi-core processors is powerful enough to run both real-time and non-real-time tasks of ROS. Implementing ROS based on a system with a multi-core processor can help reduce communication overhead by replacing inter-system communication with inter-core communication, decrease the system cost and simplify the system design. However, it is still an open issue for how to make ROS be real-time on multi-core processors.

There are challenges to make ROS be real-time on multi-core processors. First, considering the portability, a real-time OS environment should be provided to support the execution of real-time ROS tasks. It is challenging to run both a real-time OS and a general-purpose OS without interfering each other on multi-core processors. In particular, it is not trivial to provide a mechanism so interrupts, devices and other hardware resources can be separated

* Corresponding author.
E-mail addresses: weihongxing@buaa.edu.cn (H. Wei), guanyxxxy@263.net (Y. Guan), tan@utk.edu (J. Tan), cszlshao@comp.polyu.edu.hk (Z. Shao).
[1] Member, IEEE.

http://dx.doi.org/10.1016/j.future.2015.05.008
0167-739X/

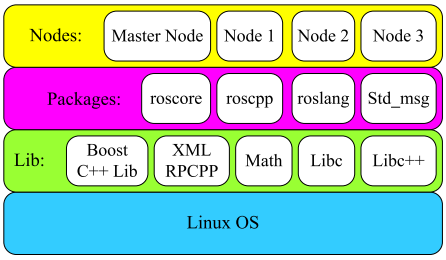


Fig. 1. The software architecture of ROS.

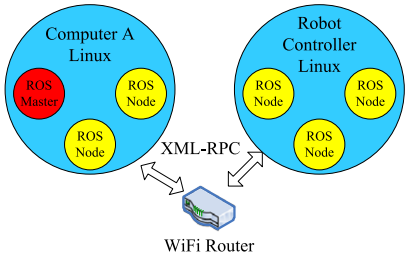


Fig. 2. The remote control model of ROS.

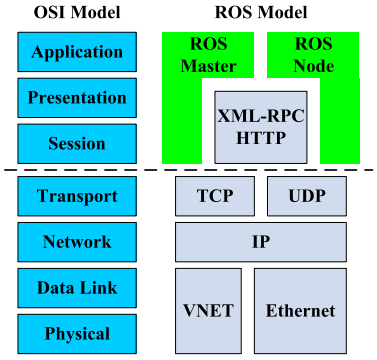


Fig. 3. The communication model of ROS and its corresponding OSI model.

and isolated to support real-time and non-real-time ROS tasks. Second, we also need to provide effective and efficient communication mechanisms between real-time and non-real-time tasks.

This paper for the first time presents a real-time ROS architecture called RT-ROS on multi-core processors. RT-ROS provides an integrated real-time/non-real-time task execution environment so real-time and non-real-time ROS nodes can be separately run on a real-time OS and Linux, respectively, with different processor cores. In such a way, real-time tasks can be supported by real-time ROS nodes on a real-time OS, while non-real-time ROS nodes on Linux can provide other functions of ROS. In RT-ROS, we develop a hybrid OS platform that can support the execution of real-time and non-real-time OSes. RT-ROS provides a mechanism in its hybrid OS so processor cores and other hardware resources such as interrupts and devices are divided and isolated, by which real-time ROS tasks on a real-time OS and non-real-time ROS tasks on a general-purpose OS can be run separately without interfering each other. Furthermore, in RT-ROS, we develop an efficient communication mechanism between real-time and non-real-time OSes.

We have implemented RT-ROS on an Intel dual-core processor. Nuttx [5], an open source real-time OS, is chosen as the real-time platform, and Linux is used as the general-purpose OS. We further implement the RT-ROS system in the industrial controller of a 6-DOF modular manipulator. The experimental results show that RT-ROS can effectively provide real-time support for the ROS platform with high performance by exploring the multi-core architecture.

The remainder of this paper is organized as follows. Section 2 introduces the background. Sections 3 and 4 present the design and implementation of RT-ROS, respectively. Evaluation is presented is Section 5. Section 6 introduces the related work. Finally, the conclusions are drawn in Section 7.

2. Background

In this section, we provide the background. We first introduce ROS and then Nuttx that is used as the real-time OS in the implementation of RT-ROS.

2.1. ROS

ROS (Robot Operating System) is an open-source and reusable software platform providing libraries, tools and conventions that can help to create high-performance robot applications quickly and easily. It provides standardized interfaces for hardware control, tools for creating, debugging, distributing and running procedures, and libraries for developing programs. So far, about 500 packages have been made available in ROS from approximately 30 institutions [6–8].

ROS is a modular software platform installed on the Linux operating system. It contains a number of software modules encapsulated as nodes, including the master node and the functional nodes. Fig. 1 gives the software architecture of ROS, which consists of libraries, packages and nodes installed on Linux. Such a modular architecture supports distributed network communication. Like the router in a LAN, the master node administrates and monitors the running of the functional nodes and their peer-to-peer communications.

Fig. 2 shows a typical remote control model of ROS. The ROS nodes in a robot are connected to these in the remote computer through the WIFI router. The communications among nodes are realized by XML/RPC (Remote Procedure Call) based on the TCP/IP protocol.

Fig. 3 illustrates the ROS communication model and its corresponding OSI model. It can be seen that an application layer based on the XML-RPC HTTP protocol is constructed on the TCP/IP architecture. Therefore, messages transmitted among nodes are not data packages but webpage files based on the http protocol.

2.2. Nuttx

In order to construct the hybrid operating system, we need a portable real-time operating system architecture. There are many real-time operating systems such as VxWorks, QNX, eCOS, and μcOS-II. Considering the portability of ROS, we choose Nuttx as our real-time OS platform.

Nuttx is an open-source embedded real time operating system (RTOS) developed by Gregory Nutt [5]. Nuttx supports the Posix and ANSI standards, and can be applied in microcontrollers from 8-bit to 32-bit. In addition, by adopting the standard APIs from Unix and other common RTOSes such as VxWorks, Nuttx also provides some functionalities not available under the Posix and ANSI standards. Compared with other real-time operating systems such as VxWorks, Windows CE and μC/OS-II, Nuttx has the following advantages:

(1) Various system services. Nuttx provides a number of system supports such as the UIP protocol stack, networking, device drivers and virtue file system, which are useful for application development.

(2) Small footprint. Nuttx only requires very little memory. For example, 4 MB memory is enough to run Nuttx.

(3) Easy extension. It is convenient to extend Nuttx to new processor architectures such as SoC architectures. This makes it possible to port real-time ROS nodes among different multi-core processors.

III. RT-ROS

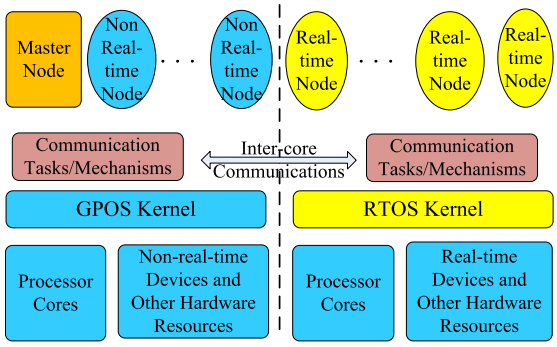


Fig. 4. The system architecture of RT-ROS.

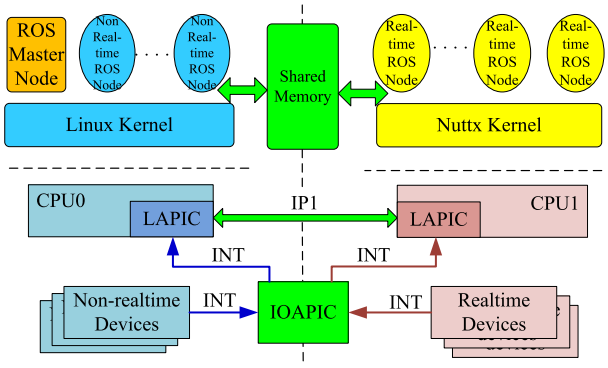


Fig. 5. The implementation of RT-ROS on an Intel dual-core processor.

Therefore, in the implementation of RT-ROS, real-time ROS nodes are run as real-time tasks in Nuttx.

3. RT-ROS

The system architecture of RT-ROS is shown in Fig. 4. At the application layer, ROS nodes are divided into real-time and non-real-time nodes. ROS nodes can communicate with each other via either inner-core or inter-core communications. These cores do communications via shared memory. At the OS level, GPOS (General-Purpose Operating System) and RTOS reside on isolated hardware resources such as processor cores, interrupts, and devices. At the hardware level, hardware resources are divided so non-real-time ROS nodes and real-time ROS nodes are separately run based on GPOS or RTOS, respectively.

4. System implementation

We have implemented RT-ROS on an Intel dual-core processor with the IA32 SMP architectures [9] as shown in Fig. 5. Intel dual-core processors are selected because they are with powerful computing capacity and complete software support. Note that RT-ROS is a general architecture and other multi-core processors can be adopted as well.

In the implementation, Linux is used as the GPOS and Nuttx as the RTOS. RT-ROS consists of two parts. One part is the non-real-time ROS system on Linux, and the other part is the real-time ROS system on Nuttx. Each part has its own CPU, memory, interrupts and peripheral devices. Accordingly, there are two kinds of ROS nodes, i.e., the real-time nodes and the non-real-time ones. The communications among nodes are performed by XML/RPC under the TCP/IP protocol. Because real-time ROS nodes and non-real-time ROS nodes run in different CPUs, VNET, that is a virtual network interface and interacts with the underlying data by sharing memory, acts as the channel for their communications.

The hardware isolation is implemented based on the APIC architecture [9] in the IA32 SMP architecture. With the APIC architecture, each CPU has a local APIC to receive interrupt message generated by an I/O APIC which collects and manages interrupt signals sent by I/O devices. Every local APIC has a unique ID so that we can program an I/O APIC to deliver a specific interrupt to a specific CPU thus to a specific OS. This is used to implement a hardware interrupt dispatcher for distributing interrupts to different OSes.

To run two OSes on two cores, we use CPU-hotplug [10]. CPU-hotplug is a feature in Linux kernel which can be used to dynamically enable or disable a CPU core. This can be done by calling two kernel API functions, namely, cpu_up() and cpu_down(). When cpu_down() is called with a CPU core number as the parameter, processes and interrupts of that CPU core will be migrated to other CPUs and the CPU core will go to sleep until cpu_up() is called.

To boot two OSes, we utilize IPI (Inter-Processor Interrupt) [9] in the IA32 SMP architecture. An IPI is generated by one CPU's local APIC and handled by another CPU as a normal interrupt. There is a special IPI called INIT IPI. If one CPU has received an INIT IPI, it will perform a boot process from the address passed by INIT IPI. This can be used by CPU hotplug for booting another OS. Furthermore, a general IPI is sent with an interrupt vector number where the CPU receiving that IPI will handle it as a normal interrupt. The general IPI is used to communicate among CPUs and OSes.

We develop a kernel model which can load the Nuttx image into memory and boot it. The boot process is as follows. First, Linux is booted normally. Then, the Nuttx-load kernel module is loaded into the Linux kernel, and it can start the Nuttx monitor. The Nuttx monitor loads the Nuttx image into memory, calls cpu_down() to disable a CPU core and issues an INIT IPI with the entry address of the Nuttx image to boot the Nuttx on that CPU core. Through the Nuttx monitor, the Nuttx can also be shut down or restarted dynamically.

In the implementation, devices are divided into real-time and non-real-time ones. Non-real-time devices are controlled by Linux and have normal Linux device drivers. Real-time devices are controlled by Nuttx and have Nuttx device drivers. Linux drivers for real-time devices are configured as disabled. Real-time device drivers set the I/O APIC to route interrupts to Nuttx. When booting Nuttx, Linux will pass the IRQ (Interrupt ReQuest) table (a table records each device's IRQ number) as the parameter. It will be used by Nuttx device drivers to allocate IRQ numbers so as to avoid IRQ conflicts. If two different devices happen to share the same IRQ number, which is common for PCI devices [11], the non-real-time one has to be disabled. However, if hardware supports MSI (Message Signaled Interrupts) [11], a device can be programmed to deliver interrupts directly to the corresponding CPU core by bypassing the I/O APIC, so the problem can be solved.

We use the shared memory of the SMP architecture to implement a bidirectional message FIFO (First In First Out). One of the biggest problems in communications is to synchronize two tasks between Linux and Nuttx. We have implemented a spinlock module that can be used to synchronize two tasks in Linux and Nuttx, respectively.

Several changes have been made to support the development of real-time ROS nodes on Nuttx. In order to develop real-time ROS nodes such as the RTroscpp nodes, the C++ support needs to be added in Nuttx. We add the light-weight library μClibc++ into Nuttx for the purpose of developing light-weight RTOS tasks. The floating-point math library in the standard libc has been provided in Nuttx, due to the fact that the floating-point calculation is always needed in real-time robot control. The http protocol has been ported in Nuttx, so http webpage files can be transferred among ROS nodes. The drivers of the real-time PCI, Ethercat and CAN bus devices are also added to Nuttx, since each real-time bus has a separate protocol framework.

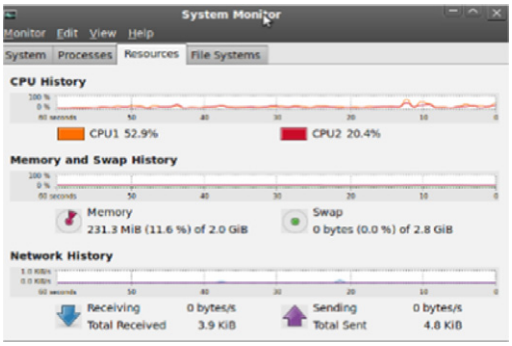


Fig. 6. CPU occupancy rates in case 1.

Table 1
The meanings of abbreviations.

Abbreviation	Meaning
DOF	Degree of freedom
RVIZ	ROS visualization
RPC	Remote procedure call

5. Evaluation

An X86-based industrial robot controller is employed to test the performance of RT-ROS, including the isolation between GPOS and RTOS, and the execution efficiency of the whole system. The testing platform consists of the following hardware: (1) Mainboard: AIMB-780; (2) CPU: Intel(R) Pentium (R) Dual CPU E2200 of 2.2 GHz; (3) Memory: DDR2 of 2 GB; (4) CAN device: Advantech PCI-1680U CAN; (5) VGA card: HD6670 DDR5 of 1 GB. The software of the testing platform includes: (1) Ubuntu 11.10 with Linux Kernel 3.2; (2) ROS; (3) OpenGL v1.4; (4) Our RT-ROS; (5) Nuttx v6.26 with μClibc++, math lib and real-time device drivers; (6) Matlab 2010a. OpenGL v1.4 is selected as this is the newest version supported by our platform. Some abbreviations used are shown in Table 1.

5.1. CPU occupancy testing

Figs. 6 and 7 illustrate the CPU occupancy rates with different cases. The CPU occupation of the dual-core processor is tested and compared in two cases. In the first case, we run both non-real-time and real-time ROS nodes in Linux. The CPU occupation results obtained by the system resource manager are shown in Fig. 6. It indicates that the GPOS tasks are shared by both CPUs. The occupancy rate of CPU1 is 52.9%, and that of CPU2 is 20.4%. In the second case, we run the non-real-time ROS nodes in Linux and the real-time ROS nodes in Nuttx at the same time. The system resource manager in Linux then displays that the occupancy rate of CPU2 becomes zero as shown in Fig. 7. This indicates that CPU2 is totally occupied by the Nuttx real-time system. Therefore, GPOS and RTOS can run simultaneously on different processor cores.

5.2. Real-time interrupt response time

An important metric to evaluate the performance of the real-time system is the interrupt response time. To test it accurately, we run a single real-time ROS node in Nuttx, which handles the external timer interrupt. Two different cases are used to show the effects of the non-real-time nodes on the real-time interrupt response time. In case A, the real-time ROS node runs in Nuttx when only the ROS master node is running simultaneously in Linux. In case B, a 3D RVIZ simulation program is simultaneously run on both the real-time ROS node and Linux. A Matlab script is written to show the interrupt response time dynamically.

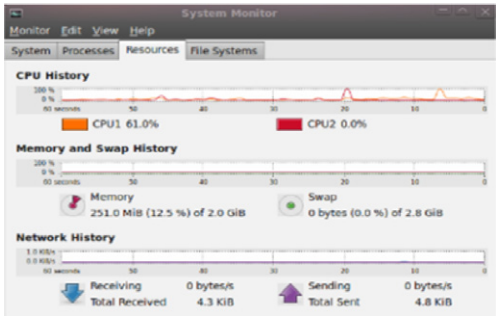


Fig. 7. CPU occupancy rates in case 2.

Table 2
The average communication time.

Communication types	Time
Non-real-time node ↔ Non-real-time node	>1 ms
Non-real-time node ↔ Real-time node	0.1–1 ms
Real-time node ↔ Real-time node	<100 ns

The experimental results are shown in Fig. 8, in which the upper straight line denotes the maximum interrupt response time and the lower one indicates the minimum one. These two figures illustrate that the interrupt response time does not affected by the non-real-time system. It can be observed that the interrupt response times in both cases lie in the same interval from 1250 to 2250 ns, implying that the effect of the non-real-time system on the real-time one is negligible. This again verifies the isolation of the two subsystems.

5.3. Communication time

We also measure the communication time between different ROS nodes. The communication time between non-real-time ROS nodes can be tested by using the standard ROS tool—rosping, and that between two real-time ROS nodes is measured by the system timer. In the experiment, only a single real-time ROS node is running in Nuttx, and only the rosping and the non-real-time ROS node for message printing are running in Linux. Because the communications among ROS nodes are made by the mechanism of the XML-RPC call under the TCP protocol, the communication time varies a lot.

The average communication times for three different kinds of node communications are shown in Table 2. It can be seen that the average communication time between two non-real-time nodes is longer than 1 ms, that between a non-real-time node and a real-time one is from 0.1 to 1 ms, and that between two real-time nodes is less than 100 ns. Therefore, the short communication time between real-time ROS nodes show that RT-ROS can be utilize for real-time robot control.

5.4. RT-ROS for controlling a 6-DOF modular manipulator

To conduct real experiments on robots, we apply RT-ROS to control a 6-DOF modular manipulator. Fig. 9 shows the software architecture. This figure illustrates how different devices communicate and interact with each other under the real-time and non-real-time OS environments. For the convenience of communication and control, an ID number is assigned to each module of the manipulator. When the controller emits an instruction containing an ID number, only the module having the same ID number will receive and respond to it. If the ID number of an instruction is 255 (i.e., the hexadecimal 0xFF), it is a broadcast instruction that will be received by all the robotic modules. RT-ROS is installed into both the manipulator controller and a remote computer, which are connected

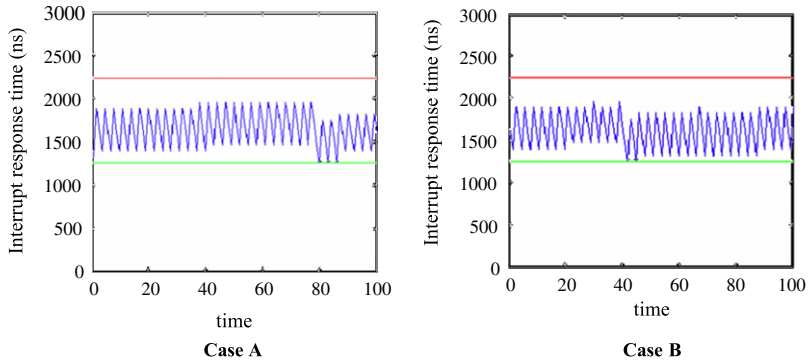


Fig. 8. The real-time interrupt response time (ns).

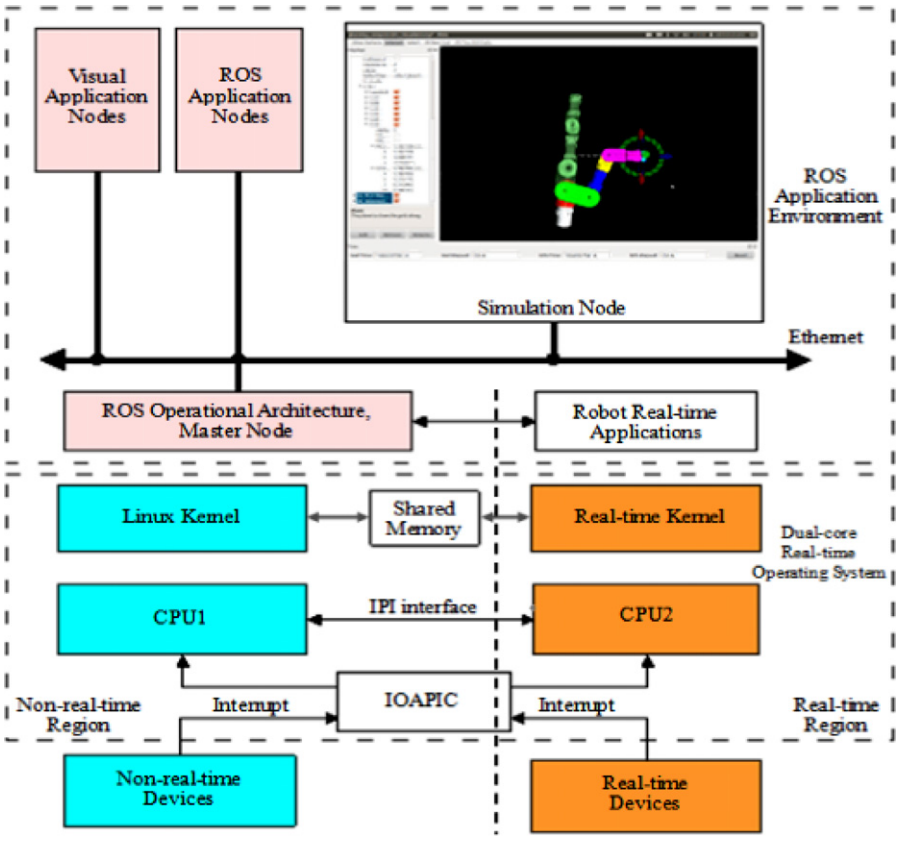


Fig. 9. The software architecture of the modular.

through Ethernet. The real-time control is realized by the real-time ROS nodes in Nuttx, while the remote manipulation and online learning have been performed by the non-real-time ROS nodes in Linux.

Fig. 10 shows the overall control architecture of the manipulator. The online learning is visualized in the ROS 3D demonstration platform—RVIZ. The model of the manipulator can be directly loaded from Solidworks into RVIZ to perform 3D simulation. In RVIZ, the end module of the 3D simulation model can be dragged by using the mouse to carry out the online teaching. As shown in Fig. 11, the path planning can also be realized for the manipulator in such a 3D simulation way.

The real-time ROS system performs real-time operations on the manipulator, including the closed-loop motion control, the kinematics solution, the status inquiry and the error handling. During the operation process of the manipulator, any one of the angular velocity, angular displacement (or position) and electric current can be used to control the rotation of the joints. Here, the position-based method is employed, so the corresponding angular displacement information is emitted to every joint to limit its motion range. It is found by motion control experiments that synchronous movement among the joints cannot be well guaranteed by the position-based method directly. Besides the method itself, there are several other factors affecting the motion synchronism, such as the different reduction gear ratios of the joints and the friction. Theoretically speaking, reducing the friction, adjusting the reduction gear ratios, or even replacing the position-based method by the angular velocity or electric current based method, can be helpful to realize the motion synchronism. However, all these methods are difficult to implement in practice. Instead, we try a simpler and more feasible way. In our implementation, a refined kinematics ROS node is developed to perform high-frequency interpolation calculation to yield more accurate control information for the angular displacement.

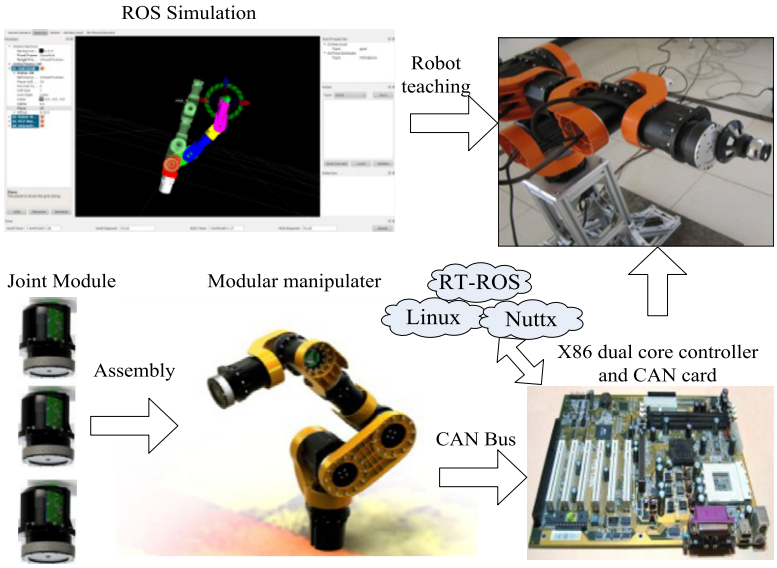


Fig. 10. The overall control architecture of the manipulator.

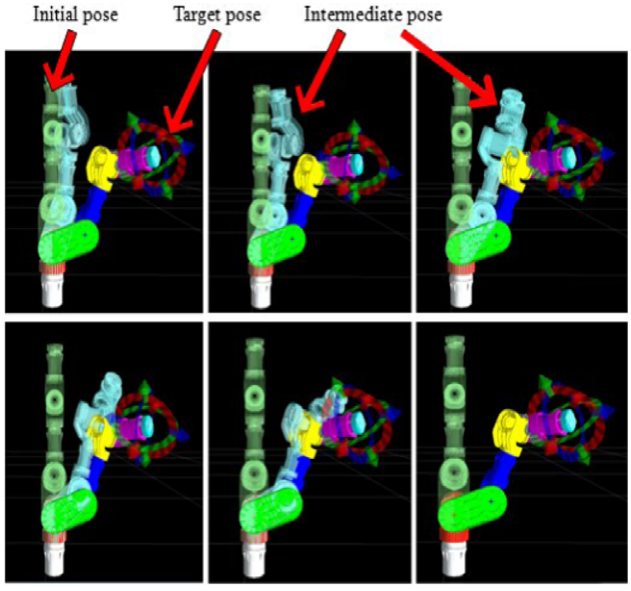


Fig. 11. Path planning in RVIZ.

Table 3
The time for the joints to rotate one degree angle.

Joints	Time (ms)	
	The position-based control method	The refined interpolation calculation method
1	350	110
2	330	108
3	331	110
4	341	109
5	320	100
6	310	105

As shown in Table 3, the motion synchronism of the six joints is well guaranteed by our method. The experimental results in Table 3 show that the times for rotating one degree with different numbers of joints are relatively stable but still have some derivations. For example, it is 109 ms with 4 joints and 110 ms with 3 joints. These deviations are caused by the physical differences of different joints.

6. Related work

There are two major issues for the development of robotics. The first is about the enhancement of the development efficiency of the controlling software. Most existing industrial robots are designed to perform special tasks in specific environments [12,13]. Because different robots generally have different controlling software, developers always have to spend a lot of time writing diverse programs with similar functions to meet new operational requirements. Thus, the complexity of programming remains as one of the major hurdles [14]. The other problem is about the enhancement of the operational efficiency of robot control. Such efficiency mainly depends on the execution speed of the controlling software.

An effective way to solve the first problem is to use the modular software architecture. By dividing an application into a number of mutually decoupled units, the complexity of the software development can be reduced. Moreover, designers only need to update the corresponding software modules, when changing the functionalities or adding features to the robots. Because most of the modules are reusable and portable for similar operational tasks, the adaptation of the controlling software can be accomplished quickly. Some modular software frameworks have been successfully applied in the controlling software of robots such as CARMEN [15], Player [16], LCM [17], and YARP [18]. Some surveys on robot middleware and robot development environment are given in [19,20].

As mentioned above, the operational efficiency of industrial robots is mainly determined by the running speed of the controlling software. There are several ways to speed up the software system. First, a number of microcontroller modules can be connected together to form a message-based [21] or event-based [22] distributed control system, in which the controller nodes communicate with each other through a field bus [22]. Second, the real-time operating system and the non-real-time one can be combined together to construct a hybrid one [23], which can handle both real-time and non-real-time tasks simultaneously by separating the corresponding interrupts automatically. Third, the multi-core processors can be employed [24], and the real-time and non-real-time software can run in different CPUs of the same processor [25,26]. Conventional systems are generally based on two-level single-core processors with the low-level processor for the real-time software and the high-level one for the non-real-time software [27]. Because a considerable amount of time is consumed by the frequent communication between different processors, the operational efficiency of the whole system is inevitably degraded. Multi-core processors can help solve this problem.

7. Conclusions and future work

In this paper, we proposed a real-time ROS architecture called RT-RTOS on multi-core processors. RT-ROS provides an integrated real-time/non-real-time task execution environment so real-time and non-real-time ROS nodes can be separately run on a real-time OS and Linux, respectively, with different processor cores. We have implemented RT-ROS on a dual-core processor and conducted various experiments with real robot applications. The experimental results show that RT-ROS can effectively provide real-time support for the ROS platform with high performance by exploring the multi-core architecture.

There are several directions for the future work. First, more testing and improvement will be conducted to enhance the performance of the software system. For example, the angular velocity or electric current based method can be adopted to get a more accurate synchronous motion control. Moreover, many emerging memory and storage techniques have been proposed [28–47]. We will study how to utilize them to further optimize the performance of RT-ROS. Finally, the proposed architecture is general and can support different types of general-purpose and real-time OSes. In the future, we will port other real-time kernels on the proposed architecture.

Acknowledgments

This work was supported by the National High Technology Research and Development Program of China ("863" Program) (2012AA041402 and 2012AA041405), National Natural Science Foundation of China (Grant No. 61175079 and No. 51105012), and National Key Technology Support Program (Grant No. 2013BAH45F01). A preliminary version of this work appears in the Proceedings of the 2011 International Conference on Robotics and Biomimetics [48].

References

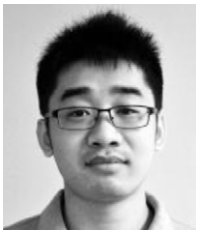
[1] ROS, 2015. http://www.ros.org/.
[2] J.M. Romano, J.P. Brindza, K.J. Kuchenbecker, ROS open-source audio recognizer: ROAR environmental sound detection tools for robot programming, Auton. Robots 34 (3) (2013) 207–215.
[3] B. Hannaford, J. Rosen, D.W. Friedman, et al., Raven-II: an open platform for surgical robotics research, IEEE Trans. Biomed. Eng. 60 (4) (2013) 954–959.
[4] P. Bouchier, Embedded ROS, IEEE Robot. Autom. Mag. 18 (3) (2013) 17–19.
[5] NuttX, 2015 http://www.nuttx.org/.
[6] S. Cousins, Exponential growth of ROS, IEEE Robot. Autom. Mag. 18 (1) (2011) 19–20.
[7] S. Cousins, B. Gerkey, K. Conley, W. Garage, Sharing software with ROS, IEEE Robot. Autom. Mag. 17 (2) (2010) 12–14.
[8] S. Cousins, ROS on the PR2, IEEE Robot. Autom. Mag. 13 (3) (2010) 23–25.
[9] Intel Corporation. Intel 64 and IA-32 architectures software developers manual, 2009.
[10] Z. Mwaikambo, A. Raj, Linux kernel hotplug CPU support, in: Proceedings of the Linux Symposium, 2004.
[11] PCI-SIG, PCI local bus specification revision 3.0, 2002.
[12] L. Zheng, et al., Architecture-based design and optimization of genetic algorithms on multi-and many-core systems, Future Gener. Comput. Syst. (2014) 75–91.
[13] K.M. Chao, et al., Cloud E-learning for Mechatronics: CLEM, Future Gener. Comput. Syst. (2014) 1–14.
[14] Z.X. Pan, J. Polden, N. Larkin, S.V. Duin, J. Norrish, Recent progress on programming methods for industrial robots, Robot. Comput.-Integr. Manuf. 28 (2) (2012) 87–94.
[15] M. Montemerlo, N. Roy, S. Thrun, Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (CARMEN) toolkit, in: IEEE/RSJ International Conference on Intelligent Robots and Systems, 2003, pp. 2436–2441.
[16] B.P. Gerkey, R.T. Vaughan, A. Howard, The player/stage project: tools for multi-robot and distributed sensor systems, in: Proceedings of the 11th International Conference on Advanced Robotics, 2003, pp. 317–323.
[17] A. Huang, E. Olson, D. Moore, LCM: Lightweight communications and marshalling, in: IEEE/RSJ International Conference on Intelligent Robots and Systems, 2010, pp. 4057–4062.
[18] G. Metta, P. Fitzpatrick, L. Natale, YARP: Yet another robot platform, Int. J. Adv. Robot. Syst. (2006) 43–48.
[19] A. Elkady, T.M. Sobh, Robotics middleware: A comprehensive literature survey and attribute-based bibliography, J. Robot. (2012) 1–15.
[20] J. Kramer, M. Scheutz, Development environments for autonomous mobile robots: A survey, Auton. Robots 22 (2) (2007) 101–132.
[21] C. Lee, Y.S. Xu, Message-based evaluation in scheme for high-level robot control, J. Intell. Robot. Syst. 25 (2) (1999) 109–119.
[22] X.M. Li, C.J. Yang, Y. Chen, X.D. Hu, Hybrid event based control architecture for tele-robotic systems controlled through Internet, J. Zhejiang Univ. Sci. 5 (3) (2004) 296–302.
[23] V.M.F. Santos, F.M.T. Silva, Design and low-level control of a humanoid robot using a distributed architecture approach, J. Vib. Control 12 (12) (2006) 1431–1456.
[24] S.S. Srinivasa, D. Ferguson, C.J. Helfrich, et al., HERB: a home exploring robotic butler, Auton. Robots 28 (1) (2010) 5–20.
[25] Y. Wang, et al., Overhead-aware energy optimization for real-time streaming applications on multiprocessor system-on-chip, ACM Trans. Des. Autom. Electron. Syst. (2011).
[26] Y. Wang, et al., Loop scheduling with memory access reduction subject to register constraints for DSP applications, Softw. - Pract. Exp. 44 (8) (2014) 999–1026.
[27] M. Liu, D. Liu, Y. Wang, M. Wang, Z.L. Shao, On improving real-time interrupt latencies of hybrid operating systems with two-level hardware interrupts, IEEE Trans. Comput. 60 (7) (2011) 978–991.
[28] J.T. Hu, et al. Software enabled wear-leveling for hybrid PCM main memory on embedded systems, in: IEEE Design, Automation & Test in Europe Conference & Exhibition, 2013.
[29] J.T. Hu, et al. Towards energy efficient hybrid on-chip scratch pad memory with non-volatile memory, in: IEEE Design, Automation & Test in Europe Conference & Exhibition, 2011.
[30] J.T. Hu, et al., Minimizing accumulative memory load cost on multi-core DSPs with multi-level memory, J. Syst. Archit. (2013) 389–399.
[31] J.T. Hu, et al., Scheduling to optimize cache utilization for non-volatile main memories, IEEE Trans. Comput. 55 (1) (2013).
[32] J.T. Hu, et al., Data allocation optimization for hybrid scratch pad memory with SRAM and nonvolatile memory, IEEE Trans. Very Large Scale Integr. Syst. 21 (6) (2013) 1094–1102.
[33] M.Y. Zhao, et al. SLC-enabled wear leveling for MLC PCM considering process variation, in: Proceedings of the 51st Annual Design Automation Conference on Design Automation Conference, 2014, pp. 1–6.
[34] L. Shi, et al. Retention trimming for wear reduction of flash memory storage systems, in: Proceedings of the 51st Annual Design Automation Conference on Design Automation Conference, 2014, pp. 1–6.
[35] G. Dip, et al. Multirate controller design for resource-and schedule-constrained automotive ECUs, in: IEEE Design, Automation & Test in Europe Conference & Exhibition, 2013, pp. 1123–1126.
[36] D. Liu, et al., Application-specific wear leveling for extending lifetime of phase change memory in embedded systems, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 33 (10) (2014) 1450–1462.
[37] Y. Wang, et al., A reliability-aware address mapping strategy for NAND flash memory storage systems, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 33 (11) (2014) 1623–1631.
[38] Y. Wang, et al., A reliability enhanced address mapping strategy for three-dimensional (3-D) NAND flash memory, IEEE Trans. Very Large Scale Integr. Syst. 22 (11) (2014) 2402–2410.
[39] R.H. Chen, et al. DHeating: Dispersed heating repair for self-healing nand flash memory, in: Proceedings of the 9th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2013, pp. 1–10.
[40] R.H. Chen, et al., On-demand block-level address mapping in large-scale NAND flash storage systems, IEEE Trans. Comput. (2014).
[41] Y. Wang, et al. Meta-Cure: a reliability enhancement strategy for metadata in NAND flash memory storage systems, in: Proceedings of the 49st Annual Design Automation Conference on Design Automation Conference, 2012, pp. 214–219.
[42] M. Liu, Z.L. Shao, M. Wang, H.X. Wei, T.M. Wang, Implementing hybrid operating systems with two-level hardware interrupts, in: 28th IEEE International Real-Time Systems Symposium, 2007, pp. 244–253.
[43] R.H. Chen, et al. Unified non-volatile memory and NAND fash memory architecture in smartphones, in: Proceedings of 20th Asia and South Pacific Design Automation Conference, 2015, pp. 340–345.
[44] C. Zhang, et al. Deterministic crash recovery for NAND flash based storage systems, in: Proceedings of the 51st Annual Design Automation Conference on Design Automation Conference, 2014, pp. 148:1–148:6.
[45] L. Shi, et al., Exploiting process variation for write performance improvement on NAND flash memory storage systems, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. PP (99) (2015) 1. 1.
[46] C.M. Gao, et al. Exploit asymmetric error rates of cell states to improve the performance of flash memory storage systems. in: 32nd IEEE International Conference on Computer Design, 2014, pp. 202–207.

[47] M. Huang, et al. A garbage collection aware stripping method for solid-state drives, in: 20th Asia and South Pacific Design Automation Conference, 2015, pp. 334–339.
[48] Q. Yu, H. Wei, M. Liu, et al. A novel multi-OS architecture for robot application, in: IEEE International Conference on Robotics and Biomimetics, 2011, pp. 2301–2306.

Hongxing Wei was born in the Inner Mongolia Autonomous Region, China, in 1974. He received a Ph.D. from the College of Automation, Harbin Engineering University, Harbin, China, in 2001. Since 2004, he has been an Associate Professor in the School of Mechanical Engineering and Automation, Beihang University (formerly Beijing University of Aeronautics and Astronautics), Beijing, China. His current research interests include self-assembly swarm robots, modular robotics architecture, mobile sensor networks, and embedded systems.

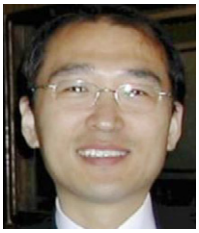
Zhenzhou Shao received the B.E. degree and M.E. degree in the Department of Information Engineering at Northeastern University, China, in 2007 and 2009, respectively, and the Ph.D. degree in The Department of Mechanical, Aerospace, and Biomedical Engineering at University of Tennessee, US, in 2013. He is currently working in the College of Information Engineering at Capital Normal University, China. His research interests include computer vision, machine learning and human-robot interaction.

Zhen Huang graduated from Beijing University of Aeronautics and Astronautics in 2010. He is a software engineer at Smokie Robotics, Inc., China.



Renhai Chen received the B.E. degree and M.E. degree in the Department of Computer Science and Technology, Shandong University, China, in 2009 and 2012, respectively. He is currently working toward the Ph.D. degree in the Department of Computing at the Hong Kong Polytechnic University. His research interests include embedded systems, mobile virtualization and hardware/software codesign, especially for secondary storage.

Yong Guan received the Ph.D. degree in College of Mechanical Electronic and Information Engineering from China University of Mining and Technology, China, in 2004. Currently, he is a Professor of Capital Normal University. His research interests include formal verification, PHM for power and embedded system design. Dr. Guan is a member of Chinese Institute of Electronics Embedded Expert Committee. He is also a member of Beijing Institute of Electronics Professional Education Committee and Standing Council Member of Beijing Society for Information Technology in Agriculture.



Jindong Tan received the PhD degree from Michigan State University, East Lansing, MI, in 2002, in Electrical and Computer Engineering. He is currently an associate professor in the Department of Electrical and Computer Engineering, Michigan Technological University. His research interests include distributed robotics, wireless sensor networks, human robot interaction, biosensing and signal processing, and surgical robots and navigation. Dr. Tan is a member of the IEEE, ACM and Sigma Xi.

Zili Shao received the B.E. degree in electronic mechanics from the University of Electronic Science and Technology of China, Sichuan, China, in 1995, and the M.S. and Ph.D. degrees from the Department of Computer Science, University of Texas at Dallas, in 2003 and 2005, respectively. He has been an Associate Professor with the Department of Computing, the Hong Kong Polytechnic University, since 2010. His research interests include embedded systems, real-time systems, compiler optimization and hardware/software co-design.