Real-Time Performance Analysis in Linux-Based Robotic Systems

Hobin Yoon, Jungmoo Song, and Jamee Lee
Advanced Software Laboratories,
Samsung Advanced Institute of Technology,
Samsung Electronics Co. Ltd.
{hobin.yoon, jmsong, jamee.lee}@samsung.com

Abstract

Mobile or humanoid robots collect environmental data and reflect back as robotic behaviors via various sensors and actuators. It is crucial this occurs within a specified time. Although real-time flavored Linux has been used to control robot arms and legs for quite a while, it has not been reported much whether the current real-time features in Linux could still meet this requirement for a much more complicated system - a humanoid with about 60 servo motors and sensors with multiple algorithms such as recognition, decision, and navigation running simultaneously. In this paper, in order to meet such requirement, adopting EtherCAT technology is introduced and its Linux implementation is illustrated. In addition, results of real-time experiments and timing analysis on a multi-core processor are presented showing Linux is a viable solution to be successfully deployed in various robotic systems.

1 Introduction

One of the key requirements of mobile or humanoid robot is precise control period. It is crucial in robot design in several ways. First it guarantees response time so that robot is able to react properly from external stimulus within a specified time. For example, when a robot hits an obstacle while it walks, if a proper re-balancing of the motion is not executed in a fraction of time, it falls down to the ground. Second, it enables smooth control of each joint which is controlled by a micro controller. Each micro controller tries to compensate movement of each servo motor if it goes too fast or slow and high jitter brings about high current consumption and even noise.

To achieve real-time communication of distributed devices, a field-bus system is used. We have selected EtherCAT over other field-bus systems for its flexible topology, simple configuration, and cost-effectiveness [18, 2]. The technology is supported and promoted by ETC (EtherCAT Technology Group) and standardized by IEC (International Electrotechnical Commission) in 2007.

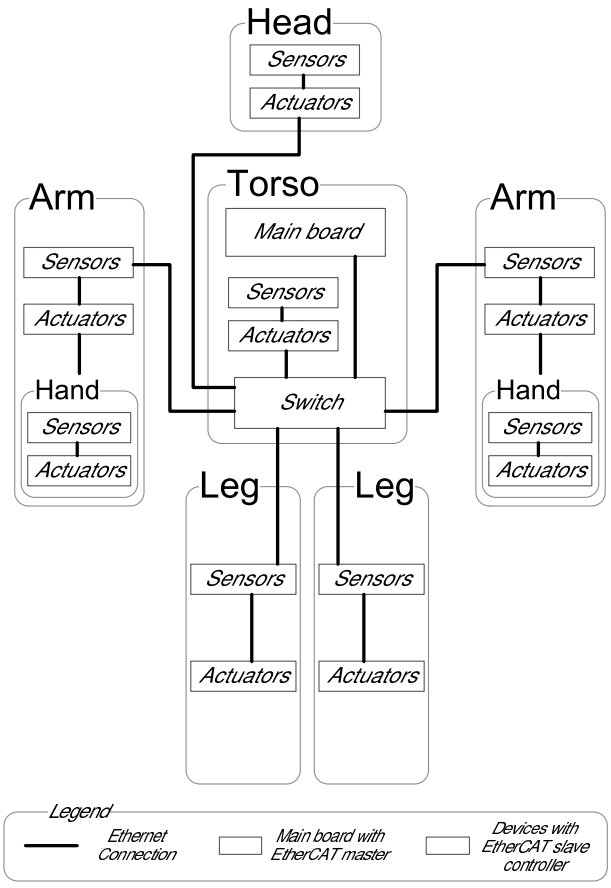


Figure 1: Deployment of EtherCAT master and slave devices

Figure 1 shows schematic diagram of EtherCAT in our robot system. Network interface card on main board in torso plays an EtherCAT master and all other rectangles represent EtherCAT slave devices. They are connected by Ethernet cables which forms various topologies such

as star, tree, and daisy chain. Each EtherCAT slave device is connected by a couple of actuators or sensors.

For EtherCAT master implementation, EtherLab was selected from other master implementations for its proper license and active community [12]. EtherCAT slave hardware is implemented by a few vendors as low-price ASIC. We have selected Beckhoff ET1100 [9].

Real-time scheduling is essential for precise control period. Traditionally, RT OSes such as QNS, RTLinux, VxWorks and Windows CE have been major players in real-time computing. Linux has been evolved a lot for the past few years in terms of real-time. Since in-kernel preemption on kernel 2.4, a lot of real-time enhancements has been added including thread-context interrupt handling, preemptible mutex, priority-inheritance mutex, high-resolution timer, user-space real-time mutex. With the help of these efforts, Linux is becoming comparable with traditional RT OSes [11, 13].

Although many efforts have been made to enhance preemption latency, there are still lots of non-preemptible critical sections and interrupt off regions. Some of the major sources of these latencies are disk IO and network IO [11].

There are several clock sources on x86 architecture such as PIT, ACPI PM, HPET, TSC. The most reliable counter with highest priority is chosen on Linux kernel start-up. TSC is usually chosen for its highest resolution. One shortcoming of the TSC was its inability to adapt dynamic voltage scaling, however, it is solved by constant TSC.

We use a multi-core processor for better efficiency in terms of power usage. However, it has a drawback in deterministic timing. On SMP kernel, preemption latency increases as more processors are added, because they contend for shared interrupt-off region and/or preempt-off region. Affinitizing task and interrupt handling can reduce preemption latency to some extent [11]. To deal more with real-time, some robotic systems use multiple OSes and boards to separate real-time task and non real-time task, although it adds more complexity and power consumption to the system [8, 20].

Tuning real-time application is dependent on application model and often underlying hardware, so it requires a lot of experiments. We followed good real-time programming guides [4, 17, 19] and the experimental result will be presented.

This paper is organized as follows. Section 2 describes design and implementation of RCKS (Robot Control Kernel Subsystem). Section 3 presents real-time performance analysis of our robotic system. Section 4 addresses further tunings. Finally, Section 5 presents concluding remarks.

2 Design and Implementation of RCKS

Figure 2 shows software architecture of our robotic system which especially details in kernel components. At the bottom of the layer, modified Ethernet device driver communicates with NIC. It has been modified to fetch received packets without interrupt. As a controlling task is guaranteed to be executed periodically, interrupts from network device driver were considered redundant. EtherCAT master which is layered on top of NIC does EtherCAT protocol handling and monitoring slave device status [12]. ECCI (EtherCAT Control Interface) is implemented on top of EtherCAT master as an interface to user-space applications.

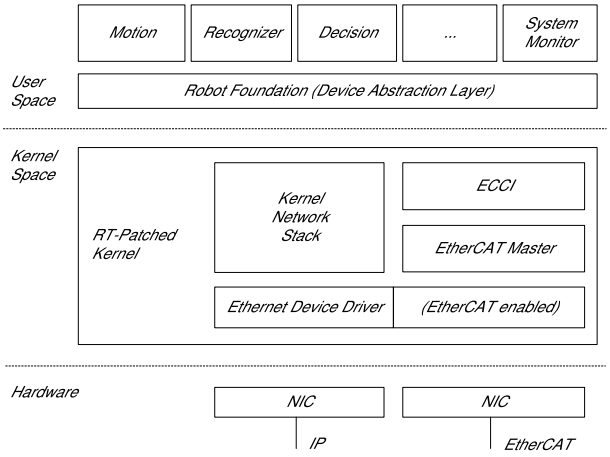


Figure 2: Software architecture of Robot Control Kernel Subsystem

We started from adopting EtherCAT master implementation. EtherLab is implemented in kernel space for two reasons. One is to avoid mode switching between kernel-space and user-space and the other is to communicate directly with network device driver. The driving application is also implemented as a kernel module [12]. Its design is optimized for performance, thereby suits for relatively small applications. However, our robot - as a humanoid robot - needs complex application logic and uses many user-space libraries that makes it inevitable to implement these in user-space. ECCI was

implemented to provide interface to user-space applications while keeping EtherCAT protocol handling intact in kernel-space. The interface includes configuring slaves, controlling actuators, reading sensor data and notifying slave status changes. It also presents a proc file system interface for exporting timing statistics.

In every cycle, ECCI receives one read-write request from the real-time task, Motion Controller. Asynchronously to this request, several non real-time tasks make read requests to ECCI. ECCI internally maintains cache of cyclic data obtained from EtherCAT master for efficiency and controls concurrent accesses from multiple tasks using a mutex which is enabled by FUTEX or PREEMPT_RT. To prevent long waiting of real-time task, ECCI employs RT-mutex [5]. RT-mutex supports priority inheritance and priority queuing which help our real-time task wait at most 1 non real-time task as shown in Figure 3.

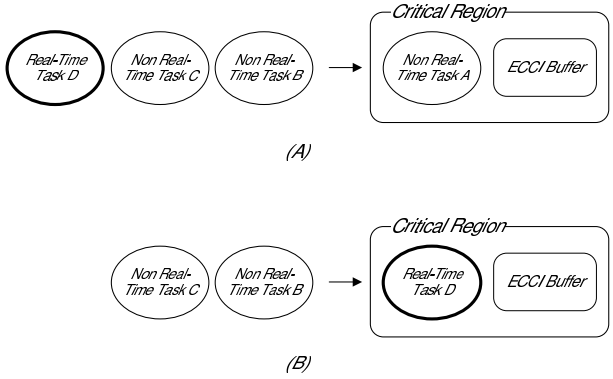


Figure 3: Concurrency control of ECCI buffer by RT-mutex: (A) Real-time task D arrives after non real-time task B and C. (B) Real-time task D acquires lock before non real-time task B and C.

Data flow in each cycle is depicted in Figure 4. The Motion Controller process sends commands which traverse through several layers to reach each actuator. Similarly, each sensor's data go through the layers to get to the Motion Controller. Data flow starts from the Motion Controller process. The process issues a read-write command which, in turn, fetches sensor data from the NIC's (Network Interface Card) buffer and composes and sends actuator commands. The sensor data have been ready at the NIC's buffer in previous cycle. The actuator commands are packetized in Ethernet frame which traverses through all slave devices. EtherCAT master returns immediately without waiting for the Ethernet frame and the thread of execution returns back to the Motion Controller. Now, the Motion Controller computes next cycle's motion plan and goes to sleep to keep steady control period. EtherCAT datagram which has been encapsulated in Ethernet packet is updated as it passes through the ESC (EtherCAT Slave Controller) on each slave device. The ESC generates interrupt to the micro controller which fetches new data from ESC's internal memory, controls actuators, gathers sensor data, and updates ESC's memory.

Sensor data take 1.5 to 2.5 cycles to reach to the Motion Controller depending on the time of occurrence. Command from the Motion Controller takes 1.5 cycles to be delivered to each actuator. Therefore it takes 3 to 4 cycles until our robot reacts to an external event—that is, 3 to 4 ms.

3 Real-Time Performance Analysis

The accuracy of the Motion Controller's control period depends on the accuracy of the sleep time in Figure 4. If the motion planning consumes reasonable amount of time, optimizing control period is essentially similar to optimizing the preemption latency of Linux kernel, and general real-time performance tunings can be applied.

3.1 General Real-Time Tunings

Linux kernel provides several tuning knobs for real-time applications. We applied some of the typical real-time tunings to achieve deterministic timing of the Motion Controller.

First, the Motion Controller process should have the highest real-time priority. It sleeps at the end of every cycle to keep constant control period which makes the task being moved from run queue to wait queue in Linux kernel. When the time expires it comes back to run queue. After that, when Linux scheduler examines the run queue, our Motion Controller should be on the highest priority run queue. Linux system call sched_setscheduler() provides this facility.

Second, dedicating one CPU for the Motion Controller is desirable. CPU shielding is a strategy in multiprocessor system which dedicates one CPU to a real-time task and other CPUs to non real-time tasks and interrupt handlers. This is beneficial to the Motion Controller for two reasons. First, it prevents latencies caused by a non real-time task or interrupt handler. They

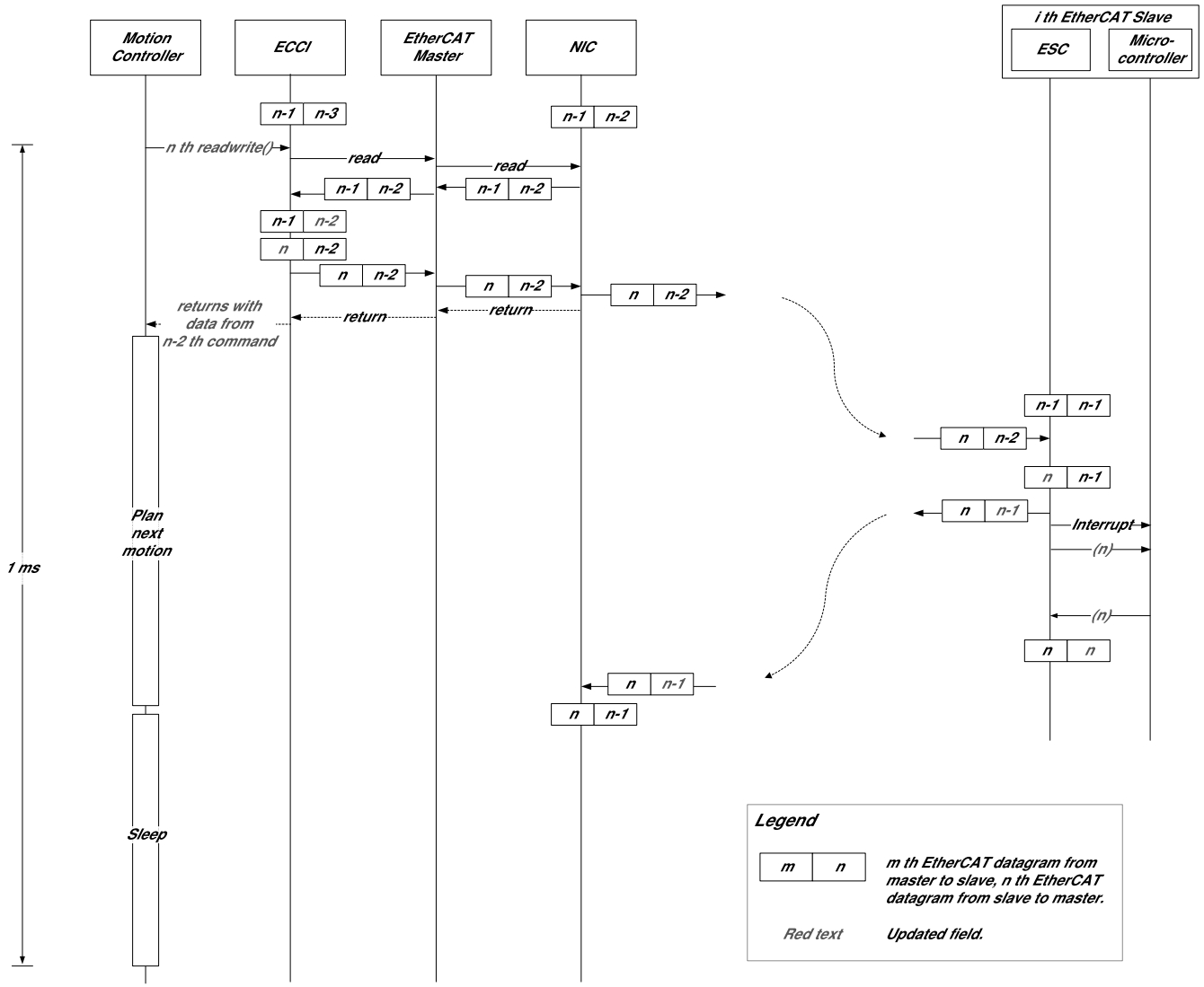


Figure 4: Data flow in a cycle. Data is exchanged through several layers.

may be in an interrupt-off and/or preemption-off region when the Motion Controller is about to be executed, thereby increasing latency. Second, high cache coherence - high coherence of instruction cache, data cache, and TLB (Translation Lookaside Buffer) - helps fast execution of the Motion Controller. Linux provides system call sched_setaffinity() for setting CPU affinity of a process. For interrupt affinity, proc file system interface /proc/irq/<irq_number>/smp_affinity and kernel API set_ioapic_affinity_irq() are provided. taskset is also a useful tool to get and set a process's CPU affinity from a shell. In addition, kernel can be configured to support CPUSETS which constrains the CPU and memory placement of tasks [1]. It is desirable to setup a resource management policy and enforce it using a global resource manager from which all child processes inherits the policy.

Last, the Motion Controller should not be paged-out to prevent high cost of fetching the page from swap area. Linux provides system call mlock() for locking the process's virtual address space into RAM.

3.2 Spinning nanosleep

RTLinux provides TIMER_ADVANCE option in its clock_nanosleep() API to enhance accuracy of sleep time. This mechanism wakes a task up before its deadline has arrived and puts it into a busy-wait loop until the deadline has arrived. This busy-wait loop improves latency for real-time task, but the process is in a busy-wait loop while waiting for the deadline [6, 10]. We

implemented this idea in Linux and applied to the Motion Controller. Since our robotic system dedicate one CPU core to the Motion Controller exclusively, spinning in the Motion Controller doesn't affect performance of other tasks. One shortcoming of this busy-waiting is increased power consumption in the core. However, this increase would be small enough compared to the overall power consumption of the robot where most power is consumed by the actuators of the joints. One word of caution is that spinning nanosleep should really sleep for some time or yield CPU to other higher or equal priority tasks before spinning on CPU. Otherwise it causes the starvation of other important kernel threads like watchdog, migration and timer thread which can lead to abnormal system behavior.

3.3 Experiment Planning

For the robot to move smoothly, it is important for the Motion Controller to send command to ECCI at the exact time of each cycle. We measured the time and generated statistics. The idea is similar to the latency analysis of cyclictest or realfeel [21, 7, 3]. Ideal period of between each consecutive time should be 1 ms; however, in practice, before the Motion Controller wakes up from sleep, the kernel may be in a critical section, which results in an additional wake-up delay of the Motion Controller.

To verify the effect of real-time tunings to the Motion Controller, we tested with all the combinations of the following options.

- Highest Priority
- CPU Shielding
- Memory Locking

In addition, spinning nanosleep is tested with all the above options turned on. Maximum spinning time is set to 50 us. When more sleep time is requested, it first nanosleep()s until 50 us remains and spins for the remaining time.

To ensure real-time performance of an operational system, it is advised to keep system load under 50% [16]. Nevertheless measuring performance under heavy load is important to observe worst-case performance. Figure 5 shows the test script which generates extremely high disk IO and network IO [14]. Figure 6 describes test environment.

- Linux kernel version: 2.6.26.8-rt16
- CPU: x86 2.4GHz Quad-core
- RAM: 2GBytes
- Robot slave devices: 59 sensors and actuators

Figure 6: Test Environment

3.4 Experimental Results

Figure 7 and 8 shows test results on unloaded system and on heavily loaded system respectively. Each combination of test was performed for 10 minutes.

Without any real-time tuning, measured maximum control period was 1,482 us on unloaded system and 346,148 us on heavily loaded system. The latter was intolerable in our robotic system.

With general real-time tunings applied—with maximum priority, memory locking and CPU shielding set—unloaded system showed average 1,017.60 us and maximum 1,044 us control period and heavily loaded system showed average 1,006.12 us and maximum 1,100 us control period.

Memory locking showed little improvement compared with other real-time tunings. It is assumed that the little improvement was due to the enough physical memory —which is 2GBytes—on our test environment which might not cause many page fault and paging-out.

Spinning nanosleep, in addition to general real-time tunings, showed best real-time performance. Control period was average 1,002.77 us and maximum 1,020 us on unloaded system and average 1,002.11 us and maximum 1,071 us on heavily loaded system which are satisfactory for smooth motion control of our robot.

Maximum values are highly unpredictable and may vary from experiment to experiment, which is because predicting longest kernel path—nested interrupt-off and preemption-off regions—is nearly impossible on heavily loaded system. For example, if the test duration is too short, the order of performance can differ from Figure 7 and 8, however when distribution is considered—like from low 99% range to low 99.999% range—we could conclude that the results are easily reproducible.

```
while true; do dd if=/dev/zero of=bigfile bs=1024000 count=1024; done &
while true; do killall hackbench; sleep 5; done &
while true; do $HACK_BENCH 20; done &
ping -l 100000 -s 10 -f localhost &
while true; do du -s / > /dev/null 2>&1 ; done &
```

Figure 5: Stress on Testing

4 Further Enhancements

Further fine-tunings are possible depending on system. In real-time systems, the ext2 file system is recommended if journaling is not required. Runlevel should be set to multi-user mode without the graphical interface to avoid additional load. Out-of-memory killer could be customized to select victim process which is least significant in terms of a robotic system. Proper tuning of sched_nr_migrate parameter is desired to limit the number of task that will move at a time [4, 17, 19].

Delayed locking technique can be applied to our system which execute a real-time task at a predefined interval [15]. This technique allows a non real-time task to enter a critical section only if the operation does not disturb the future execution of the real-time application.

5 Conclusions

Our robotic system needed a real-time OS for deterministic control of actuators and sensors, and, at the same time needed a general OS for running many processes simultaneously and using rich user-space libraries. Linux with complete in-kernel preemption patch was selected to meet the requirements and our robot system had the benefit of exploiting plentiful open-source software available in Linux.

Robot control kernel subsystem is implemented using EtherLab, an EtherCAT master implementation to control various sensors and actuators in real-time. Robot-specific application logic is implemented in user-space, while EtherCAT-specific protocol handling stays in kernel-space so the kernel can be robust from user-space bugs.

On our system, 1,000 us control period was met most of the time, with average 1,002.11 us and maximum 1,071 us on heavily loaded system which was good enough for smooth motion control. As our robotic system is still in development, we expect to get better real-time performance with further fine tunings.

References

[1] CPUSETS. Linux kernel documentation: `kernel/Documentation/cpusets.txt`.

[2] EtherCAT Technical Introduction and Overview. `http://www.packagingdigest.com/contents/pdf/EtherCAT_Introduction_en.pd%f`.

[3] Linux Real Time Patch Review - Vanilla vs. RT patch comparison. `http://www.captain.at/howto-linux-real-time-patch.php`.

[4] Real-Time Linux Wiki. Project site: `http://rt.wiki.kernel.org`.

[5] RT-mutex subsystem with PI support. Linux kernel documentation: `kernel/Documentation/rt-mutex.txt`.

[6] RTLinuxPro CPU Reservation Technology. `http://www.linuxdevices.com/articles/AT7665542109.html`.

[7] Andrew Webber. Realfeel Test of the Preemptible Kernel Patch. `http://www.linuxjournal.com/article/6405`.

[8] Berthold Bäuml and Gerd Hirzinger. When hard realtime matters: Software for complex mechatronic systems. *Robotics and Autonomous Systems*, 56(1):5–13, 2008.

[9] Beckhoff. *Hardware Data Sheet ET1100 EtherCAT Slave Controller*, Jan 2008.

[10] Cort Dougan and Zwane Mwaikambo. Lies, Misdirection, and Real-Time Measurements. `http://www.ddj.com/cpp/184401780`.

[11] S. Dietrich and D. Walker. The evolution of real-time linux. In *Proceedings of Seventh Real-Time Linux Workshop*, Nov 2005.

[12] EtherLab. *IgH EtherCAT Master 1.4.0 Preliminary Documentation*, Feb 2009.

[13] Thomas Gleixner and Douglas Niehaus. Hrtimers and beyond: Transforming the linux time subsystems. In *Ottawa Linux Symposium*, 2006.

[14] Ingo Molnar. dohell script. Linux kernel mailing list: `http://lkml.org/lkml/2005/6/22/347`.

[15] Jupyung Lee and Kyu-Ho Park. Delayed locking technique for improving real-time performance of embedded linux by prediction of timer interrupt. In *Real Time and Embedded Technology and Applications Symposium, 2005. RTAS 2005. 11th IEEE*, pages 487–496, March 2005.

[16] Paul E McKenny. 'real time' vs 'real fast': How to choose? In *Ottawa Linux Symposium*, 2008.

[17] Montavista. *Real-time Application Programmer's Guide*, 2008.

[18] S. Potra and G. Sebestyen. Ethercat protocol implementation issues on an embedded linux platform. In *Automation, Quality and Testing, Robotics, 2006 IEEE International Conference on*, volume 1, pages 420–425, May 2006.

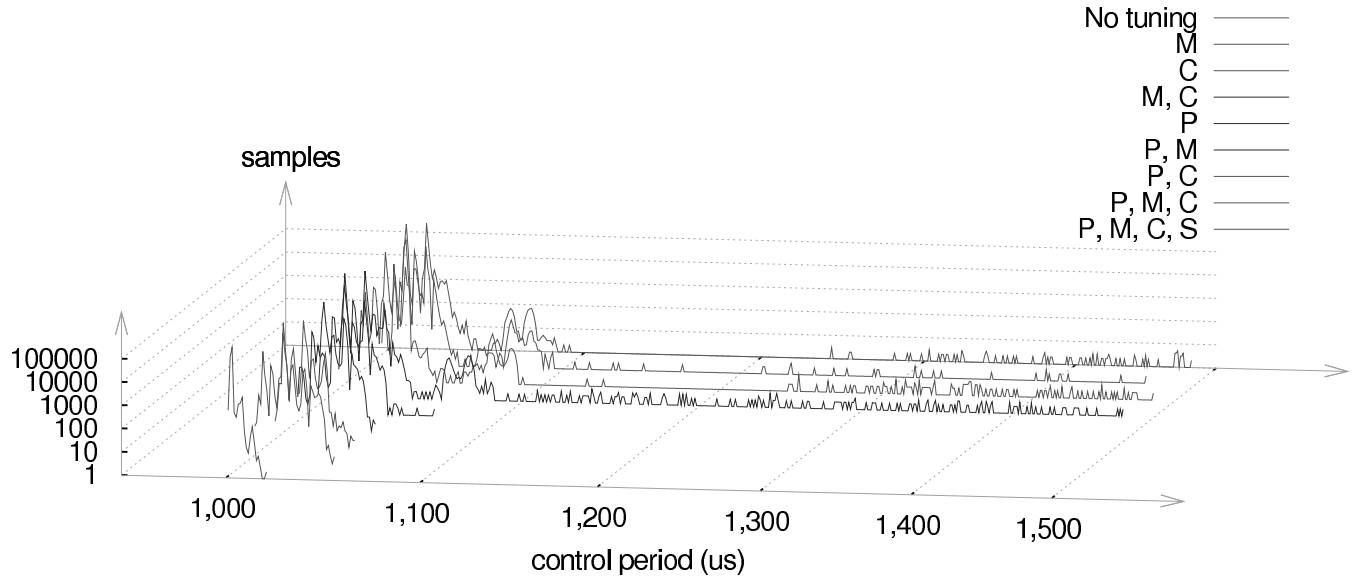
[19] Redhat. *Red Hat Enterprise MRG 1.1 Realtime Tuning Guide*, 2008.

[20] R. Tellez, F. Ferro, S. Garcia, E. Gomez, E. Jorge, D. Mora, D. Pinyol, J. Oliver, O. Torres, J. Velazquez, and D. Faconti. Reem-b: An autonomous lightweight human-size humanoid robot. In *Humanoid Robots, 2008. Humanoids 2008. 8th IEEE-RAS International Conference on*, pages 462–468, Dec. 2008.

[21] Thomas Gleixsner. Cyclictest. `http://rt.wiki.kernel.org/index.php/Cyclictest`.

		No Tuning	M	C	MC	P	PM	PC	PMC	PMCS
	min	1,005.00	1,006.00	1,005.00	1,005.00	1,005.00	1,005.00	1,004.00	1,004.00	1,001.00
Low	max	1,023.00	1,032.00	1,030.00	1,028.00	1,029.00	1,030.00	1,029.00	1,032.00	1,006.00
99%	avg	1,013.45	1,020.54	1,020.01	1,014.19	1,019.46	1,019.75	1,014.48	1,017.43	1,002.72
	SD	5.02	4.94	4.51	5.10	4.03	4.37	5.34	4.21	0.48
Low	max	1,060.00	1,061.00	1,072.00	1,062.00	1,034.00	1,035.00	1,030.00	1,037.00	1,008.00
99.9%	avg	1,013.56	1,020.65	1,020.11	1,014.33	1,019.56	1,019.86	1,014.61	1,017.58	1,002.76
	SD	5.17	5.06	4.65	5.33	4.15	4.50	5.50	4.48	0.65
Low	max	1,351.00	1,085.00	1,355.00	1,331.00	1,038.00	1,038.00	1,034.00	1,039.00	1,013.00
99.99%	avg	1,013.62	1,020.70	1,020.19	1,014.41	1,019.57	1,019.88	1,014.62	1,017.60	1,002.77
	SD	5.76	5.32	5.82	6.16	4.17	4.53	5.52	4.52	0.69
Low	max	1,474.00	1,394.00	1,463.00	1,444.00	1,043.00	1,040.00	1,040.00	1,041.00	1,016.00
99.999%	avg	1,013.66	1,020.71	1,020.22	1,014.44	1,019.57	1,019.88	1,014.63	1,017.60	1,002.77
	SD	6.96	5.59	6.86	7.08	4.18	4.53	5.52	4.52	0.70
	max	1,482.00	1,464.00	1,484.00	1,477.00	1,064.00	1,044.00	1,044.00	1,044.00	1,020.00
100%	avg	1,013.67	1,020.71	1,020.23	1,014.45	1,019.58	1,019.88	1,014.63	1,017.60	1,002.77
	SD	7.12	5.74	7.02	7.23	4.18	4.53	5.52	4.52	0.70

Maximum value and statistics of control periods on unloaded system. 100% row shows distribution of entire range, while other rows show data distribution without peak values. (Unit:us, P: maximum priority, M: memory locking, C: CPU shielding, S: spinning nanosleep)

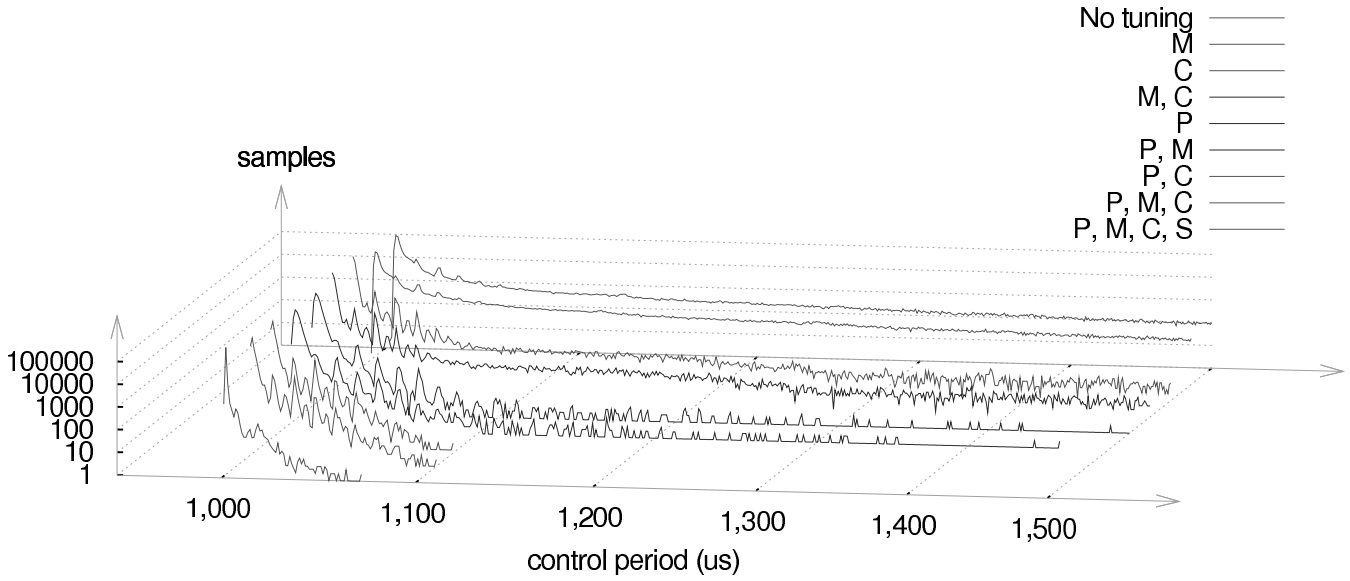


Distribution of control periods

Figure 7: Control periods on unloaded loaded system

		No Tuning	M	C	MC	P	PM	PC	PMC	PMCS
	min	1,002.00	1,003.00	1,004.00	1,004.00	1,004.00	1,004.00	1,004.00	1,004.00	1,001.00
Low	max	36,967.00	37,019.00	1,249.00	1,215.00	1,020.00	1,026.00	1,027.00	1,027.00	1,004.00
99%	avg	2,072.05	2,066.02	1,006.85	1,006.66	1,006.96	1,007.08	1,005.83	1,005.81	1,002.03
	SD	3,935.35	3,915.51	13.03	11.83	1.85	2.11	3.28	3.23	0.20
Low	max	88,376.00	87,678.00	2,004.00	2,004.00	1,050.00	1,051.00	1,048.00	1,048.00	1,019.00
99.9%	avg	2,529.41	2,521.51	1,013.85	1,013.33	1,007.17	1,007.30	1,006.09	1,006.07	1,002.08
	SD	6,314.70	6,283.76	78.55	76.08	2.90	3.25	4.29	4.26	0.74
Low	max	156,548.00	152,964.00	16,513.00	19,003.00	1,173.00	1,160.00	1,068.00	1,071.00	1,038.00
99.99%	avg	2,624.61	2,615.18	1,016.26	1,016.09	1,007.23	1,007.36	1,006.13	1,006.11	1,002.10
	SD	7,081.89	7,029.08	138.80	161.27	3.57	3.84	4.52	4.51	0.99
Low	max	236,548.00	228,477.00	39,079.00	33,098.00	1,387.00	1,319.00	1,081.00	1,088.00	1,055.00
99.999%	avg	2,640.65	2,630.67	1,018.49	1,018.29	1,007.25	1,007.38	1,006.13	1,006.12	1,002.11
	SD	7,284.39	7,219.58	280.70	285.45	4.23	4.36	4.56	4.56	1.06
	max	346,148.00	284,092.00	50,158.00	56,006.00	1,510.00	1,464.00	1,098.00	1,100.00	1,071.00
100%	avg	2,643.59	2,633.28	1,018.92	1,018.73	1,007.25	1,007.39	1,006.14	1,006.12	1,002.11
	SD	7,341.15	7,264.35	310.66	317.71	4.46	4.53	4.57	4.57	1.08

Maximum value and statistics of control periods on heavily loaded system. 100% row shows distribution of entire range, while other rows show data distribution without peak values. (Unit:us, P: maximum priority, M: memory locking, C: CPU shielding, S: spinning nanosleep)



Distribution of control periods (Values greater than 1,500 us were not depicted)

Figure 8: Control periods on heavily loaded system

Proceedings of the Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*

Programme Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
James Bottomley, *Novell*
Bdale Garbee, *HP*
Dave Jones, *Red Hat*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Alasdair Kergon, *Red Hat*
Matthew Wilson, *rPath*

Proceedings Committee

Robyn Bergeron
Chris Dukes, *workfrog.com*
Jonas Fonseca
John 'Warthog9' Hawley

With thanks to
John W. Lockhart, *Red Hat*