Proceedings of the 17th World Congress
The International Federation of Automatic Control
Seoul, Korea, July 6-11, 2008

An innovative marking machine integrated with a GNU/Linux-based embedded real-time platform ⋆

Gianfranco Ravera * **Alessio Bertone** * **Gabriele Bruzzone** ** **Massimo Caccia** **

* *Green Project S.r.l., Corso Perrone 47r, Genova, Italy(Tel: +39-0106001802; e-mail: info@greenproject.it).*
** *CNR-ISSIA, Genova, Italy (Tel: +39-01064756{57,12}; e-mail: {gabry,max}@ge.issia.cnr.it).*

Abstract: This paper discusses the integration of a stamping marking machine with a GNU/Linux-based platform for embedded real-time systems. The work, carried out in cooperation between a Small Medium Enterprise and a public research organisation, points out the possibility of adopting standard hardware and software technologies, and, in particular, free software, in the field of advanced industrial automation. Laboratory, and, in the final version of the paper, field trials, demonstrate the performance of the proposed system.

1. INTRODUCTION

The applied research presented in this paper aims at developing marking machines for continuous casting lines satisfying classical operating requirements and being ready for integration with advanced systems for automatic traceability.
Basically, marking machines for continuous casting lines have to satisfy the following operating requirements:

- fastness: the marking cycle has to be as fast as possible; for instance, in six strands application at 5-6 mpm casting speed the time available to mark each billet can be less than 10 seconds;
- flexibility: the machine has to be able to mark non only numbers and Latin characters, but also customer logos and different set of characters, e.g. Cyrillic;
- reliability: the machine must operate in very harsh environment (see, for instance Figure 1, since the billet surface is about 750C, and the environmental operating temperature can be in the range -30C - +50C).

Three classes of machines, based on different marking systems, are currently available on the market:

- stamping machines: based on the percussion of a pin (a whole character or a segment); permanent marking (the depth of the indent on the billet surface can be 2 -3 mm) with a fast marking cycle;
- powder/paint spray machines: characters are printed on the billet surface synchronizing a nozzle with the movement of an X-Y table (or attached to a robot wrist); only few characters can be printed on the frontal surface of the billet and the cycle time is higher;
- tag indent machines: a metal tag (previously generated with a laser marking system) is applied to the billet surface by a robot; the marking cycle is quite slow and the maintenance procedure are usually a little more complex than in the previous case but the result is particularly reliable for automatic reading.

Fig. 1. The *Hammer* marking machine at work in the Mechel steel plant, Russia.

Indeed, the capability of reading the billet code is currently at the centre of the competition in the marking machine market, all the competitors focusing on the billet/bloom traceability problem.
This will involve the development and integration of different machines, likely based on different, complementary technologies from automatic control to computer vision. In this context, the availability of a, possibly standard, software and hardware architecture able to support different technologies and I/O devices will dramatically ease system development, integration and maintenance, thus reducing the corresponding costs. As a consequence of the dramatic growth of computer computational power, capacity of flash memory cards, and enhancement and diffusion of the open source GNU/Linux operating system (GNU)(Lin

⋆ This work has been partially funded by Regione Liguria - Obiettivo 2 (2000-2006) Sottomisura 1.4B- L. 598/94 art. 11 "Interventi per la ricerca industriale e lo sviluppo precompetitivo" in the project "Embedded real-time platform for industrial automation and robotics".

(a)), an opportunity in this sense has been given by systems consisting of standard GNU/Linux and commercially available off-the-shelf (COTS) hardware. Indeed, in spite of its historical development as an operating system for the desktop/server environment, GNU/Linux has become quite an attractive choice in the field of embedded OSs. It provides rapid application development with a reduced time to market, thanks to its basic features of reliability, scalability, portability, open source, specifications and standards, support and large programmer base, and free of charge availability (Lennon (2001)). Its main limitation is that, having a Unix-like kernel designed to guarantee a fair sharing of resources among many users and applications, in its original version it does not satisfy real-time performance requirements, which are fundamental in the embedded operating system market.
In order to obtain real-time capabilities, many solutions have been proposed and are currently available both commercial and free, basically following two different approaches:

(1) introducing a new software layer (essentially a real-time kernel called micro/nano kernel) between Linux, which runs as a low priority process of the new real-time kernel, and the hardware, as proposed by RTLinux (RTL), RTAI (RTA) and more recently by ADEOS (ADE);
(2) applying a set of patches to a standard Linux to make it a real-time kernel, as initially proposed by the KURT project (KUR) and TymeSys Linux (Tim), and become a standard configuration option for Linux kernel starting from release 2.5.4-pre6.

The result is that GNU/Linux has become a practical option in robotics research and industry, substituting special purpose, and often expensive, hardware and proprietary real-time operating systems in a large number of applications. Examples are given by the projects MCA (Modular Controller Architecture), where a modular, network transparent and real-time capable C/C++ framework for controlling robots and other kind of hardware was developed (MCA), COMEDI (COntrol and MEasurement Device Interface), where open-source drivers, tools, and libraries for data acquisition are provided (COM), and OROCOS (Open RObot COntrol Software), whose aim is to develop a general-purpose and open robot control software package (ORO). Moreover, as discussed for example in (Wang (2002)), the trend of next generation manufacturing and factory automation systems is based on building flexible open systems using open source OSs and tools as those provided by GNU/Linux, that can easily, quickly and cost-effectively be upgraded or expanded to meet the ever-changing production requirements. OSACA (Open System Architecture for Controls within Automation systems), OMAC (Open Modular Architecture Control) and OSEC (Japan's Open System Environment Consortium) are a few examples of efforts in this direction. For up to date lists of control and robotics projects carried out using GNU/Linux the reader can refer to the RealTime Linux Foundation web site (Rea).
On the other hand, the dramatic increase in hardware performance motivated the development of systems with very strict timing requirements using standard GNU/Linux running on COTS hardware. This is, for instance, the case of the data acquisition system for nuclear physics developed at the National Super-conducting Cyclotron Laboratory of the Michigan State University (Fox et al. (2004)), where a precision of the order of a few μs was obtained by using the readout computer as an instrument rather than a general-purpose computer and utilizing a few amortization techniques of software and systems overheads.
Following this trend research carried out in cooperation between Green Project Srl, a small enterprise developing automation systems for the iron and steel industry, and the *Autonomous robotic systems and control* group of CNR-ISSIA, a public research organisation, demonstrates the possibility of developing embedded real-time robotics and manufacturing applications using a standard GNU/Linux running on commercial off-the-shelf hardware.

2. HAMMER MARKING MACHINE

The *Hammer* machine has been developed by Green Project Srl for marking of continuous casting products (e.g. billets, blooms). Its principle of operation is based on a punching pin moved orthogonally to the billet by a pneumatic cylinder (see Figure 2. Denoting with X-Y the

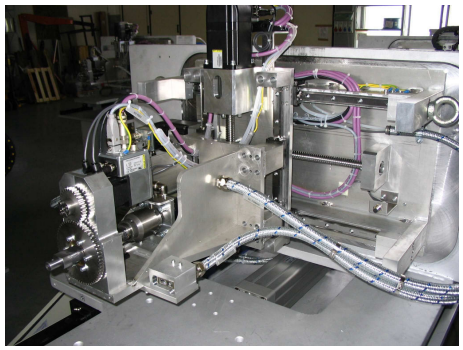

Fig. 2. View of the *Hammer* marking machine in the lab.

marked planar surface and with Z its orthogonal direction, the marking is obtained by synchronizing the motion of the puncher in a working area X-Y with the rotation around the Z axis and the puncher command. Two electrical step motors, mounted on a frame with ball screw thread and slides, move the group punch-cylinder in the X-Y plane, while a third one is used for punch rotation. In order to obtain the impact strength needed to mark the billet, the motion of the puncher along the Z axis is controlled by a pneumatic actuator piloted by electric valves. Once determined the character to be written, synchronization is performed by commanding the suitable positions of the electrical step motors and the opening/closing of the electric valves with a required time interval resolution of a couple of milliseconds. An example of marking results in typical operating conditions is shown in Figure 3. The control system is based on an industrial PC Advantech with bus ISA equipped with an Advantech PCA 6773 @ 650 MHz Intel CPU with 128 MB of RAM and one 128 MB CF. The interface with the field is handled with different technologies: time critical I/O, e.g. electro-valves, are piloted by OPTO22 modules managed by a PCL-731 Advantech industrial PC board; non time critical I/O, e.g. buttons and lamps, are piloted by Schneider Advantis

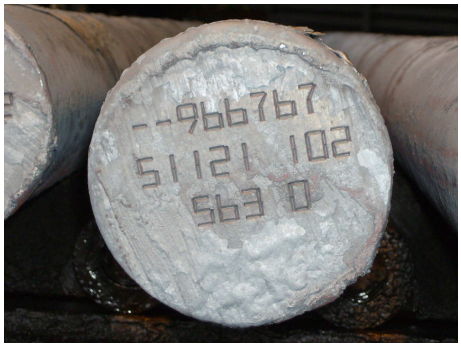

Fig. 3. Example of *Hammer* marked characters.

distributed I/O modules, while the intelligent step motors (Berger-Lahr IFS) are piloted by a CIF Hilscher board with protocol CAN Open Master through a CAN Open fieldbus. It is worth noting that the use of intelligent step motors, together with small message sizes, has allowed high timing performances (the devices can be polled with read/write frequencies of 2-3 milliseconds) even keeping the control system in a controlled environment (electrical room) at a range of about 100-200 m from the field devices. On the other hand, the integration with the plant automation network is handled by CIF Hilscher boards which can provide a standard interface for a large variety of fieldbus (Profibus, Devicenet, Control Net, etc.).
The current version of the machine has been implemented using the RTKernel real-time operating system.

3. GNU/LINUX-BASED EMBEDDED REAL-TIME PLATFORM

3.1 State-of-the-art in making GNU/Linux real-time

A real-time operating system must be able to quickly preempt any task that is currently executing when an interrupt occurs. Since it had been originally designed as a Unix-like operating system, GNU/Linux kernel does not guarantee real-time performances, i.e. a worst case interrupt response time and a deterministic execution time. The main reason of this lack of performance relies in the so-called *scheduler latency* problem, i.e. the fact that the delay between the occurrence of an interrupt and the running of the thread that is in charge of serve it can be, in particular critical situations, very long (of the order of tens of milliseconds). This problem is essentially caused by the no preemptibility of threads when running kernel code and by the presence of long critical sections of code in the kernel that cannot be interrupted.
Following the approach proposed by the Kansas University's Real-Time (KURT) Linux project in 1997 KUR and by TimeSys with the TymeSys Linux in 1998 Tim, a set of kernel patches have been introduced directly within the structure of a standard Linux in order to implement the Posix 1003.1d POS real-time extensions (high resolution timers, preemptible kernel, improved task scheduler, etc.). In particular, the so-called *preemption* Rob and *low-latency* AndIng patches, put together in a single patch and available as a standard kernel configuration option (CONFIG_PREEMPT) from Linux release 2.5.4-pre6, are based on the idea of creating opportunities for the kernel scheduler to be run more often minimizing the time between the occurrence of an event and the running of the scheduler. Basic ideas are, on one hand, modifying the spinlock macros and the interrupt return code so that, if it is safe to preempt the current process and a rescheduling request is pending, the scheduler is called, and, on the other hand, introducing explicit preemption points in blocks of code of the kernel that may execute for long stretches of time. The result is that a standard linux kernel compiled enabling the CONFIG_PREEMPT patch presents a maximum latency and jitter of the order of a few tens of msecs also when particularly demanding activities, such as accessing large amounts of memory causing page faults, stressing the system through keyboard caps-lock, console switch, non memory-mapped I/O, /proc file system access, and process fork are executed Abeni et al. (2002).

3.2 Proposed approach and implementation

Considering that in the last years the dramatic improvements in size and performance of solid-state memories, such as compact flash (CF) memory cards and RAM, have made less critical the construction of customized embedded GNU/Linux systems, an approach relying on embedding standard GNU/Linux, as discussed in Bruzzone et al. (2006), programming avoiding operations causing system latency, and using a suitable timer to generate time interrupt signals, has been implemented. In the following a detailed discussion of the reasons of the choice of particular hardware and software tools, together with the presentation of a set of C++ classes designed in order to simplify and standardize the development of control system applications, will be given.

COTS hardware The selected hardware components are standard industrial PC-derived computers and relevant I/O boards. In particular, CPU boards supplied with both a PC/104+ and a PCI bus (i.e. supporting I/O cards for both bus types) are used. These standard hardware components give the developer the opportunity of finding, at a low cost, a high number of I/O boards and CPUs with very high computing power. Moreover, within the chipset of PC-derived computers is always present a real-time clock (RTC) that, suitably programmed, can be used as a good time basis for the control application.

Real Time Clock (RTC) as a time basis for real-time Since the kernel native timing resolution is rather poor (usually 10 milliseconds on the current versions), high frequency time signals can be generated by using the RTC, a particular clock usually built into the chipset of all PCs and PC-derived computers. The main function of the RTC is to keep the date and time while the computer is turned off. However it can also be programmed to generate interrupt signals from a slow 2 Hz frequency to a relatively fast 8192 Hz one, in increments of powers of two. From Linux point of view, the RTC is seen as a particular read only character device */dev/rtc*, on which a user thread can execute a blocking read. The interrupt frequency, starting and stopping of the RTC can be easily programmed into the RTC via various *ioctl* calls.

POSIX threads Real-time data acquisition and control modules are implemented as POSIX threads (in the fol-

lowing called *pthreads* for short). GNU/Linux provides two different libraries to manage threads which give the user a set of primitives for creating, running, stopping and resuming *pthreads*, synchronizing their operations, and specifying their scheduling policies and priorities: LinuxThreads (Lin (b)) and NPTL (NPT). LinuxThreads was the first library implementing POSIX threads available for Linux but it had a number of issues with true POSIX compliance, particularly in the areas of signal handling, scheduling, and inter-process synchronization primitives. On the other side, the original implementation of NPTL mostly missed the real-time support: although available, system calls to select real-time scheduling had no effect. Actually, NPTL developers didn't implement real-time due to the lack of efficient synchronization mechanisms in the kernel (NPT). However, after recent enhancements of Linux, in particular the introduction of futexes (Fast Userspace Locking system calls) (Franke et al. (2002)), nowadays NPTL supports real-time and provides a complete and efficient implementation of POSIX threads for Linux.
Anyway, using either LinuxThreads or NPTL, there are three possible types of scheduling policies for *pthreads*: SCHED_FIFO, SCHED_RR and SCHED_OTHER and the priorities range from 0 (lowest) to 99 (highest). Linux manages *pthreads* having SCHED_FIFO scheduling policy in a fully preemptive and priority-based way, guaranteeing that at any given instant the highest priority ready SCHED_FIFO *pthread* is the one that is executed by the CPU. SCHED_RR *pthreads* are treated by the scheduler in a similar way but with a round-robin algorithm, and are rarely used. Normal Linux threads are *pthreads* having priority 0 (the lowest) and SCHED_OTHER scheduling policy, i.e. a Unix-like time-sharing one. Being the priority of SCHED_FIFO pthreads always greater than 0 they never can be preempted by a Linux thread. Thus it is possible to implement real-time applications delegating time-critical tasks that cannot be interrupted to *pthreads* having SCHED_FIFO scheduling policy (referred in the following as real-time *pthreads*), while SCHED_OTHER Linux threads can be active only when there are no real-time *pthreads* running or ready to run.

Custom real-time scheduler and thread, data and communication management In order to simplify and standardize the development of control system applications, a set of C++ classes encapsulating real-time and Linux threads, a custom scheduler devoted to manage their timing requests and message queues for inter-thread communications have been implemented, in a first phase using LinuxThreads library primitives, and in a second phase using NPTL. This was allowed by recent enhancements of Linux, in particular the introduction of futexes (fast userspace mutexes) Franke et al. (2002), which enabled NPTL to support realtime and provides a complete and efficient implementation of pthreads for Linux. The taxonomy of the thread classes in shown in Figure 4, where the abstract base class (ABC) Thread embodies the main characteristics of a generic thread and acts as a base class on which other classes can be built. The Thread class derives *SchedFIFOThread* and *LinuxThread* classes, representing thread with scheduling policy and priority (SCHED_FIFO, 1...99) and (SCHED_OTHER, 0) respectively, i.e. real-time *pthreads* and normal Linux threads. It is worth noting that the class *SchedFIFOThread* implements a generic real-time *pthread* that is usually waiting for a specific event.

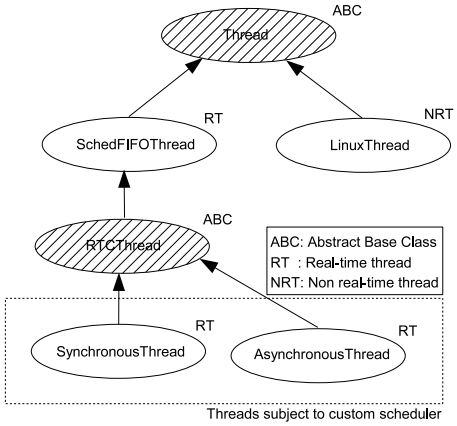

Fig. 4. Thread classes taxonomy.

The main characteristics of a generic real-time *pthread* subject to the custom scheduler management are embodied by the abstract base class *RTCThread* class, derived from *SchedFIFOThread*. *RTCThread* derives the *SynchronousThread* and *AsynchronousThread* classes representing respectively real-time *pthreads* that need to be periodically scheduled at a requested frequency and real-time *pthreads* that need to be awaken after a desired time interval.
The custom scheduler is simply a real-time *pthread* having the highest available priority that is cyclically activated by an interrupt generated by the RTC at a prefixed frequency and whose aim is to manage requests from *AsynchronousThreads* and *SynchronousThreads*.
A few other classes, e.g. message queues that are not supported by the LinuxThreads library, were designed and implemented to provide in an easy way inter-thread and network communications.

4. CONTROL SYSTEM ARCHITECTURE

The *Hammer* control system architecture, shown in Figure 5, basically consists of the *Machine* thread, a finite state machine (a kind of virtual PLC) handling digital sensor I/O including the electro-valves controlling the pneumatic puncher, and the *Head* thread, a finite state machine controlling the writing of the characters through the management of the intelligent step motors connected to the CAN Open fieldbus. For hardware optimization, the four digital outputs needed for piloting the puncher electrovalves are included in an eight channel digital output port together with signal piloting conventional devices. Thus, they are written by the *Machine* thread, on the basis of commands received by the *Head* thread through a suitable communication queue. Data exchange between the host field board (Profibus DP) and the application machine is managed by the *Host* thread, while the *HMI* (Human Machine Interface) thread handles communications with the operator interface. The *Profibus* and the *CANOpen* threads manage respectively the communications with the Profibus and the CANOpen buses while the *cmdStreamToQueue* and the *tlmQueueToDatagram* threads handle the network interface. The *Machine* thread is implemented

as a real-time synchronous *pthread* running at 1024 Hz for a fast update of electro-valve commands, while the *Head* thread is a real-time asynchronous one. Indeed, once received from the *Machine* thread the command of writing a character, it plans the marking head motion and then executes it as a sequence of: *i*) send command to the step motors, *ii*) sleep for a few milliseconds, *iii*) check if the command has been completed. The *Host* thread is a real-time synchronous *pthread* running at 16 Hz, while the *HMI* thread is a real-time synchronous one running at 8 Hz. *Profibus*, *CANOpen*, *cmdStreamToQueue* and *tlmQueue-ToDatagram* threads are *SchedFIFOThreads*. The custom scheduler runs at 2048 Hz.

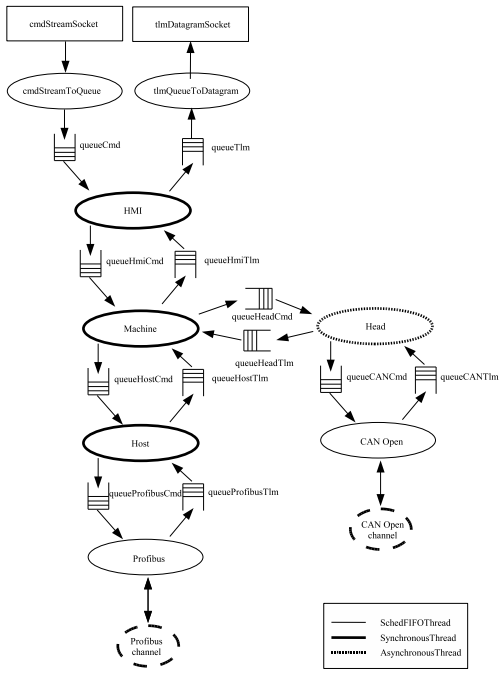

Fig. 5. Hammer machine control system architecture.

5. EXPERIMENTAL RESULTS

Experiments were carried out in the lab in order to evaluate the performance of the proposed platform. The *Profibus* and *CANOpen* threads communicates with RS-232 serial links at 115.2 Kbps simulating the behavior of the Profibus and CANOpen buses. The *Hammer* control system was run on a target machine, an Advantech PCM-9577 Single Board Computer, equipped with an Intel Pentium III CPU at 1.2 GHz, 512 MB of RAM, 4 RS-232 serial ports, one 128 MB CF card and an Ethernet link at 100 Mbps. Three PC-104 modules (Advantech PCM-3724, Advantech PCM-3718HG and Diamond Ruby-MM-1612) were used to perform digital I/O, analog input and output respectively. The overall system was put in a CF card from where the SBC was programmed to boot. A Java program running on a host computer simulated the behavior of an Ethernet-connected Human Machine Interface by sending random commands to the system at a frequency of 10 Hz and receiving the telemetry at 8 Hz from the target. Besides the *Hammer* control system threads, on the target machine there was also running *Calc*, a Linux thread that uninterruptedly calculated random numbers, whose only purpose was to computationally load the system.
The 64-bit time-stamp counter (TSC) was used as timer, independent from the RTC, to measure the application performance. It keeps an accurate count of every cycle that occurs on the processor since the machine was booted. To access this counter the RDTSC (read time-stamp counter) instruction is available. The *scheduler* thread and each thread deriving from *RTCThread* class were designed in such a way to give the user the option to record in a suitable array (kept in RAM for efficiency reasons) their timing values calculated by reading the TSC. In particular, the scheduler thread and the threads of type *SynchronousThread* record in their respective arrays the histograms of the difference between the requested ideal period and the difference between two following synchronization events (error on the period) whereas the threads of type *AsynchronousThread* save the difference between the requested delay and the actual obtained one (error on the delay). Since the *Hammer* application requires the system to be very reactive in applying puncher electro-valve commands, the time spent to transfer the corresponding message from the *Head* thread to the *Machine* thread is monitored too. At the end of the application the arrays are copied from RAM to files, thus allowing an off-line analysis of the data.
Experiments to evaluate system performances with and without CONFIG_PREEMPT option activated were carried out running the system for about two weeks using both the LinuxThreads library with Linux 2.6.9 and the NPTL 2.3.5 with Linux 2.6.21.1. In particular three timing parameters were considered:

(1) T_{max}: the maximum error on the requested period of the scheduler and of synchronous threads
(2) $T_{99.99}$: the 99.99th percentile of the scheduler error, i.e. one scheduler sample over ten thousand has an error higher than $T_{99.99}$
(3) $T_{99.9999}$: the 99.9999th percentile of the scheduler error, i.e. one scheduler sample over one million has an error higher than $T_{99.9999}$

Results, summarized in Table 1, point out how the performance of recent kernels and POSIX library NPTL is noticeably better than that obtainable using old kernels and the outdated LinuxThreads library. In particular, activating the CONFIG_PREEMPT option reduces the maximum error, while NPTL is more efficient than LinuxThreads.

Library	Preempt.	T_{max}	$T_{99.99}$	$T_{99.9999}$
LinuxThreads	no	752 μs	44 μs	114 μs
LinuxThreads	yes	281 μs	39 μs	112 μs
NPTL	no	495 μs	39 μs	114 μs
NPTL	yes	243 μs	18 μs	106 μs

Table 1. System performance evaluation.

As far as the scheduling sequences of *pthreads* are concerned, the observed behavior using NPTL is identical to that found using LinuxThreads. The results obtained using LinuxThreads, shown in the following, are hence still fully applicable using NPTL. Considering that, denoting with

T the period of the custom scheduler, i.e. the interval between two RTC interrupts, the *Machine* thread has a period of $2T$ and a priority higher than that of the *Head* thread, the possible scheduling sequences of the kernel scheduler and of the *Machine*, *Head* and custom scheduler threads are shown in Figure 6, where k, s and m denote the kernel scheduler, custom scheduler and *Machine* thread execution times respectively (assumed constant in first approximation) and W_r and W_m are the requested and the actual measured waiting time of the *Head* thread. Since the

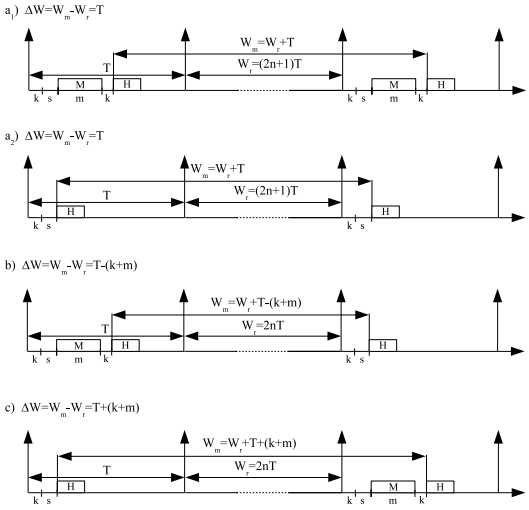

Fig. 6. Scheduling sequences showing the three possible waiting times for the *Head* thread.

system was designed to guarantee an actual waiting time W in the execution of the suspended thread not lower than W_r, the error on the waiting time, $\Delta W = W_m - W_r$ can assume the values:

a) T, when the requested waiting time is an odd multiple of T, i.e. $W_r = (2n+1)T$
b) $T-(k+m)$, when the requested waiting time is an even multiple of T, i.e. $W_r = 2nT$ and the *Head* thread sends its request to be resumed in a RTC slot where the *Machine* thread is scheduled to run.
c) $T+(k+m)$, when the requested waiting time is an even multiple of T, i.e. $W_r = 2nT$ and the *Head* thread sends its request to be resumed in a RTC slot where the *Machine* is not scheduled to run.

As expected, three peaks spaced by the time required for kernel scheduling and *Machine* thread execution are visible in Figure 7 representing the histogram of the error on the requested waiting time of the *Head* thread. In particular, the measured error peak is located at 486 μs (i.e. the scheduler period), while the two secondary peaks are situated at 379 μs and 594 μs respectively, corresponding to an execution time of 108 μs for kernel scheduler and *Machine* thread. During the tests, the requested *Head* thread waiting time was randomly extracted between $2T$ and $10T$, i.e. between about 1 and 5 ms.

6. CONCLUSION

The integration of a stamping marking machine with a GNU/Linux-based platform for embedded real-time systems has been discussed in this paper, showing how stan-

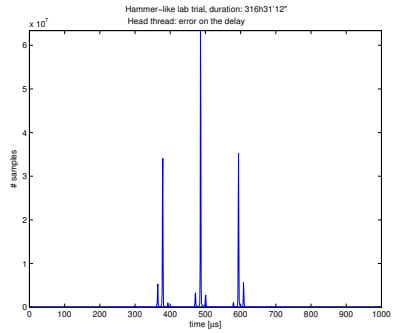

Fig. 7. Histogram of the *Head* thread error on the requested waiting time.

dard Linux-based platforms can support industrial real-time applications up to a couple of KHz. The proposed platform can easily support expected developments in the field of automatic reading of the billet codes, requiring, for instance, the integration of vision and laser scan systems.

REFERENCES

http://home.gna.org/adeos.
http://www.zipworld.com.au/~akpm/linux/schedlat.html.
http://www.comedi.org.
http://www.gnu.org.
http://people.redhat.com/~mingo.
http://www.ittc.ku.edu/kurt.
http://www.kernel.org, a.
http://pauillac.inria.fr/~xleroy/linuxthreads, b.
http://mca2.sourceforge.net.
http://people.redhat.com/drepper/nptl-design.pdf.
http://www.orocos.org.
http://www.opengroup.org.
http://www.aero.polimi.it/~rtai.
http://www.fsmlabs.com.
http://www.realtimelinuxfoundation.org.
http://www.kernel.org/pub/linux/kernel/people/rml.
http://www.timesys.com.
L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole. A measurement-based analysis of the real-time performance of Linux. In *Proc. of Eight IEEE Real-Time and Embedded Technology and Applications Symposium*, 2002.
G. Bruzzone, M. Caccia, A. Bertone, and G. Ravera. Standard Linux for embedded real-time manufacturing control systems. In *Proc. of IEEE 14th Mediterranean Conference on Control and Automation*, 2006.
R. Fox, E. Kasten, K. Orji, C. Bolen, C. Maurice, and J. Venema. Real-time results without real-time systems. *IEEE Trans. on Nuclear Science*, 51(3):571–575, 2004.
H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: fast userlevel locking in Linux. In *Ottawa Linux Symposium*, pages 479–489, 2002.
A. Lennon. Embedding Linux. *IEE Review*, 47(3):33–37, 2001.
L. Wang. Factory automation systems: evolution and trends. In *Proc. of AUTOTEST*, pages 880– 886, 2002.