Experiences with soft real-time, multithreaded control of a mobile robot*

Giuseppe Beccari Stefano Caselli Francesco Zanichelli Luigi Capuzzello
Dipartimento di Ingegneria dell'Informazione
Università di Parma - 43100 Parma - Italy
e-mail: {beccari,caselli,mczane,capu}@ce.unipr.it

Abstract

The paper reports on the experience we are gaining on the soft-real time, multithreaded control of our Nomad 200 mobile robot. Linux-based POSIX scheduling and multithreading have been evaluated by quantitative assessments and extensively exploited in the organization of Nomad control software. Fine-grained concurrency, efficient data sharing, and real preemption have proven crucial for the implementation of low-level behaviors and other higher-level tasks responsible for robot navigation. Based on the multi-threaded real-time control architecture, the mobile robot has the ability to explore, map, and navigate office environments using sonars and infrared sensors, at up to 20 cm/s navigation speed and 10 Hz sensor interpretation rate.

1 Introduction

One of the key issues in automonous robotics research concerns the definition of the logical architecture of the control software. While it is increasingly recognized the effectiveness of layered, hybrid architectures [1, 2, 3], (de)coupling deliberative competence and reactive skills, it still not obvious what programming and operating system support is best tailored for their concrete implementation. Whatever the available hardware (from simple microcontrollers up to multiprocessors), common requirements for development at all levels of the hybrid architecture include support for concurrency (activities), reliable communication and synchronization. The sequencing and the behavioral layers additionally emphasize the need for task control capabilities (suspend/resume) and guaranteed real-time execution [4, 5].

Since a few years, several commercial mobile robots with rich sensoriality have been introduced in the market. Mainly designed to serve the needs of the research community, these systems feature integrated PC-based controllers running standard UNIX-like operating systems and support user programming by means of *client-server* interaction. While those valuable environments allow for both familiar development tools and off-board distributed processing, they appear somewhat limited for those applications also requiring finer concurrency, frequent sharing of sensor data and reliable periodic task scheduling. This is often the case when users need to integrate low-level control and perceptual capabilities over those envisioned by the manufacturer.

In light of these requirements and as an useful extension to the preexisting programming environment, we have introduced and experimented a control software architecture for a Nomad mobile robot which heavily relies upon a multithreading (MT) library and the POSIX-compliant scheduler extensions [6] generally available in the Linux operating system. Threads can be regarded as a standardized, OS independent way to achieve lightweight concurrency in modern operating systems also allowing a full exploitation of symmetrical multiprocessor (SMP) hardware. Our approach tries to assess the contribution of multithreading to mobile robot control applications where the real-time component plays an important role.

Thus, a crucial question to be answered in our work relates to the extent Linux can serve as a viable operating system for soft real-time mobile robot applications. Recent Linux kernels include some of the features required by real-time operation (*e.g.* fixed priority scheduling and the disabling of memory paging), but current kernel architecture is mainly impaired by being not preemptible because of the lack of fine grain locks. A process (or thread) running in kernel mode during the execution of a system call cannot be preempted until it blocks in the kernel. Higher priority processes can thus be delayed for an interval which could be non-negligible (preemption latency). Furthermore, overall response time to external events is not optimized due to the fact that certain device drivers disable interrupts for several milliseconds. While we simply attempt to estimate the impact of these limitations on mobile robot applications, an on-going project has been reported [7] to bring hard real-time capabilities to Linux systems by means of an external real-time executive.

The remaining of the paper is organized as follows. Section 2 describes other known OS- and language-based multithreading and programming approaches to mobile robot control. Section 3 gives a short introduction to programming with threads. In section 4 the multi-threaded architecture we have devised to support the development of behavior-based and sequencing layers is presented. Next, section 5 describes how the MT support has been applied to an office navigation task for our Nomad robot and the achieved performance. Finally, a

*Partial support for this research has been provided by MURST, Italy (Project "Design Methodologies and Tools of High Performance Systems for Distributed Applications") and by C.N.R., Italy (Special Project "Real-Time Computing in the Real-World").

few remarks conclude the paper.

2 Related work

The integration of a thread library within the Nomad mobile robot control system has been reported for the University of Kansas robot at the 1996 AAAI robot competition [8]. Unfortunately, the paper does not offer any insight on the actual system architecture and performance.

Specialized robot programming languages addressing a particular level of the hybrid architecture hierarchy include threading capabilities. The *L* language [9] focuses on the development of behavior-based controllers and supports LISP-based multithreaded programming for embedded systems. The *Colbert* language [10] targets programs specifying the scheduling and control of tasks and behaviors at the bottom layer of a hybrid architecture. *Colbert* emphasizes the underlying finite state automata semantics and the C-like interactive development and debugging.

Commercial mobile platforms favor an easy start for their users through standard Linux-based client/server programming tools. The Nomadic programming environment [11] is based on a single server program (*robotd*) which dispatches basic motion commands and returns sensor state with the help of dedicated hardware controllers. As additional capabilities have to be coded in client external programs, tight integration becomes unfeasible. RWI's BeeSoft architecture [12] features a number of server programs dealing with base sensors and motion control, collision avoidance, map making and path planning. Client programs are structured in event-driven light-weight modules interacting with servers through a specialized router process (*TCX*) which is derived from the Task Control Architecture [13].

Table 1 summarizes the main characteristics of some of the aforementioned programming tools for mobile robots. Most table entries shown as N (No) may become available otherwise through operating system services operating at the process level.

3 Background on threads programming

The foremost concept behind multithreading programming is related to the separation between the address space and the thread of control [14]. Traditional operating systems promote the process abstraction which virtualizes the CPU while allowing only for one thread of control per address space. Under a multithreading library user applications can be structured fully according to their internal concurrency as a number of interacting threads, yet safely insulated from the outside by a private address mapping. A set of threads appears to be appropriate to cooperative concurrency, unlike process granularity typical of multiuser operating systems. Multithreaded applications can largely benefit from non-blocking systems calls, efficient shared memory communication and low overhead context switching.

The POSIX 1003.1c standard specifies the interfaces to thread creation, activation and cancellation, scheduling, as well as synchronization through mutexes, condition variables and semphores [15]. Several POSIX-compliant thread packages (*pthreads*) are available for most Unix platforms whereas Solaris and NT offer proprietary, albeit functionally similar, libraries. We have implemented the multithreaded control system by means of LinuxThreads, a multithreading library designed for the Linux operating system [16]. LinuxThreads is a robust and efficient kernel-based library in which threads are created by the *clone* system call and scheduled like regular processes on the available processors.

4 Multithreaded control architecture

The main aim of the control architecture is to provide reliable and flexible support to the execution and interaction of behaviors and other control activities as well as to their sequencing and monitoring.

The key idea is that behaviors and other tightly interacting activities are mapped to the fine grain concurrency of threads within a single Linux process. Low level robot control and loosely coupled processing modules (possibly distributed over the LAN) are encapsulated into other processes accessed via standard client-server socket interaction. Our effort builds over the existing manufacturer supplied low level software, thus preserving optimized access and exploitation of robot hardware.

The control software organization is depicted in Fig. 1. It exploits the LinuxThreads package and Linux POSIX.1b scheduler to provide a basic framework of periodic threads with:

- timing handled by the *nanosleep* system call;
- time-faults handled by user selectable policies;
- premptive scheduling policies with static priorities;

Each thread requiring periodic execution simply invokes the *nanosleep* delaying system call and is thus responsible for its own activation or overrun detection. As this method (based on self-scheduling of threads) provides no scheduling guarantee to the designer, a more complex solution based on a event scheduler is under development. Currently, the MT control system supports a set of generic services and user-defined policies for dealing with scheduling overruns (time faults). According to the relevance or role of the thread missing its deadline, this event can be either ignored or traced or handled by a user-supplied function as it is done in the Chimera II operating system [17, 18]. These mechanisms allow the definition of integrated systems able to redistribute available computing resources among deliberative and real-time activities as a function of environment changes [19].

All real-time threads are given a static priority and assigned to one of the real-time policies (*SCHED_FIFO, SCHED_RR*) known to the POSIX scheduler. Both strategies allow real preemption among threads and other processes.

A few thread-based behaviors general enough to be included in many navigation task have also been defined. Specifically, a thread is devoted to the high frequency periodic acquisition of the robot sensor state (proprioceptive and sonar, infrared, and bumper readings) while other threads are always active and enforce safety: they cause the robot to stop when obstacles are detected inside a predefined safety belt.

	Nomadic env.	Beesoft	Colbert
Concurrency: process-based	Y	Y	N
light-weight	N	N	Y
Real-time scheduling: soft	N	N	N
hard	N	N	N
Preemption	N	N	N
IPC/syncronization	Y	Y	Y
Shared memory	N	N	Y
Client-server	Y	Y	N

Table 1: Software tools for mobile robot programming.

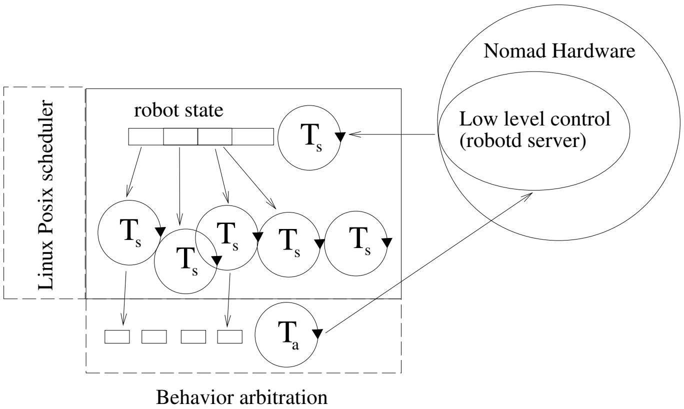

Figure 1: The organization of the thread-based control software.

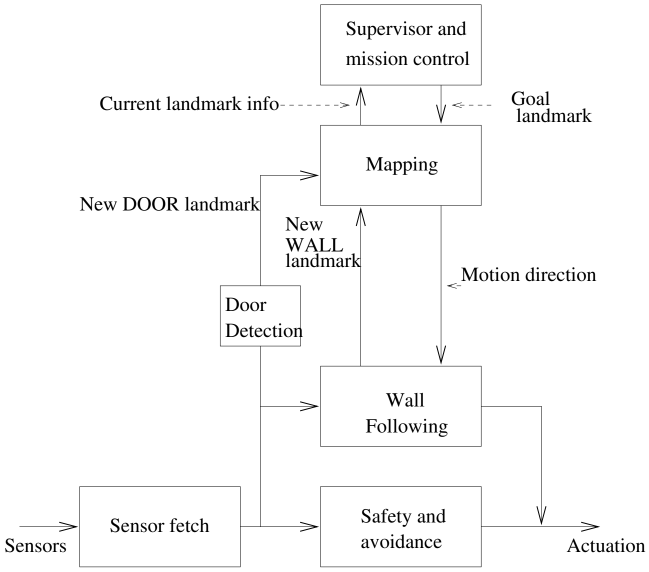

Figure 2: MT control architecture for indoor navigation.

4.1 Performance tests

A number of tests has been performed on the Linux system of a Nomad 200 mobile base available in our laboratory, in order to assess the efficiency of task handling and interprocess communication. All reported times refer to a Linux 2.0.24 system on an Intel 486 DX/2 processor with 256k cache and 16MB of memory. Table 2 highlights how even on a uniprocessor system the overhead of threads-based concurrency is appreciably smaller than in the case of the traditional process-based approach, as much as four times for shared memory versus TCP/IP communication.

Under heavy system activity, current Linux kernels cannot guarantee zero-latency scheduling for real-time (higher priority) processes. We ran simple test processes at 20 Hz together with another intensive time-sharing task and measured the distribution of scheduling latency using the same methodology of [20]. The results of this investigation are shown in Table 3. On the average, these results indicate that a scheduling delay less than 10 ms is obtained about 99% of the times when the process is scheduled with the real-time `SCHED_FIFO` policy. Regular time-sharing processes achieve significantly larger latencies. A commercial SYSVR4 Unix with solid real-time performance like Solaris 2.X improves the result for real-time processes to 100 %. Similar latency distributions were obtained in other tests where we applied Rate Monothonic priority assignment [21] to two *SCHED_FIFO* processes running at 20 Hz and 10Hz, respectively. From a practical viewpoint, these results appear compatible with the soft real-time requirements of most mobile robot applications.

5 Application to a navigation task

We have evaluated the multithreading support into the software control architecture used for indoor navigation experiments of Julius Caesar (JC), the Nomad 200 mobile robot available in our laboratory.

Based on pre-existing structural landmarks, JC has the ability to explore, map, and navigate one among several office buildings of our site sharing similar structural features, while coping with slow environment variations and local dynamics. The mobile robot develops and maintains an internal representation of the environment in terms of a topological and qualitative map augmented with metric and orientation attributes [22].

The types of structural features suitable as navigation landmarks largely depend upon the available robot sensoriality. In this particular application, we have developed a specific perception module to detect office doors (either open or closed), whereas walls are identified by

	threads (avg/max)	processes (avg/max)
Context switch	20/22	28/31
Creation time	1680/1729	3022/3065
IPC via socket	n/a	868/950
IPC via shared memory w/ mutex	198/265	n/a

Table 2: Threads *vs.* processes efficiency comparison (all times in microseconds).

Latency	0	1-10	10-20	21-30	31-40	41-50	higher than 50 msec.
`SCHED_FIFO` process	95.7	4.2	0.1	0.01	0.01	0.01	0.01
`SCHED_OTHER` process	89.5	5.2	5.32	0.1	0.01	0.01	0.1

Table 3: Comparison of percentual distribution of scheduling latencies (in ms) for Linux `SCHED_FIFO` real-time and `SCHED_OTHER` time-sharing processes (50 ms nominal period).

continuous monitoring and clustering robot proprioceptive status as induced by the boundary following behavior. All other entities of significant extent that survive behavioral filtering are classified as "unknown landmark".

The multi-threaded real-time control architecture significantly contributes to the adequate navigation performance we achieved, up to 20 cm/s robot speed and a consistent 100 millisecond sensor interpretation rate. Predictable, safe navigation inside standard, people inhabited office environments has been accomplished by means of the control architecture depicted in Fig. 2. Each unit has been implemented on a separate thread within a single Linux process. Periodic threads are all scheduled at 10 Hz in the real-time round-robin policy *SCHED_RR*. Sensor data is obtained from *robotd* server by the highest priority *Sensor Fetch* thread and shared with other threads. High update rates enable *Sensor Fetch* to perform sensor preprocessing (*e.g.* running averages and minima) to improve robustness against noise. The *Safety* behavior runs at the same frequency of *Sensor Fetch* and provides safe navigation. Motor behavior *Wall Following* identifies wall segments as a by-product of the boundary following. *Door Detect* examines sonar and IR readings to determine the presence of doors. The higher-level *Mapping* thread interacts with perceptual units and with *Mission Control* synchronizing its activity on a single condition variable set by other threads.

At this stage, the overall navigation task is simply specified into a *Mission Control* thread responsible of behavior sequencing and task progress monitoring like in the following:

```
load_conf(DEFAULT);

activate_thread(fetch_sensor);

activate_behavior(safety);
activate_behavior(door_detect);
activate_behavior(wall_following);
activate_behavior(mapping);

/* Rare example of an elegant GOTO  */
goto(LANDMARK_6);  /*  */

/* Max allowable time to destination  */
t1= timer_event(180);

/* Wait for an useful event  */
event= waiton_event_queue();

switch(event)
{
case GOALREACHED:
  disable_behavior(wall_following);
  timer_clear(t1);

  if(curr_landmark.type == OPEN_DOOR)
    {
    enter_door();  /* motion primitive */
    say("You've got mail !");
    end_mission();
    }

  else
    goto(LANDMARK_12);
  break;

case TIMEOUT:
  ...
/* reduce_speed
and try to self localize again */
  break;

case DANGER:   /* from safety behavior !  */
  ...
  break;
}
```

System events, timeouts and messages from underlying active threads are made known to the *Mission Control* thread through a single event queue. When a thread causes activation overruns, a specific event may be generated to the purpose of changing its frequency. In the navigation experiments we performed, only the *Wall Following* thread sporadically missed its deadline due to slow execution of motion requests by the *robotd* server process. TCP/IP-based interconnection and unreactive operation of *robotd* appear to decrease the overall performance of the whole multithreaded control architecture and alternative arrangements are being considered.

6 Conclusions

The paper described the multithreaded support we devised for the control of a Nomad 200 mobile robot. Layered upon Linux POSIX-compliant real-time scheduler and the LinuxThreads package, the MT support allows the development of tighly integrated control software within the usual Linux environment. Further work will address scheduler extensions to support other real-time scheduling policies and a more efficient integration with Nomadic *robotd* server.

References

[1] Connell J., "SSS: A hybrid architecture applied to robot navigation", IEEE Int. Conf. on Robotics and Automation, Nice, France, May 1992.

[2] Arkin R.C., "Integrating behavioral, perceptual and world knowledge in reactive navigation", Robotics and Autonomous Systems, 6:105-122, 1990.

[3] Konolige K., Myers K., Saffiotti A., Ruspini E., "The Saphira architecture: a design for autonomy", J. of Experimental and Theoretical Artificial Intelligence, 9:215-235, 1997.

[4] Gat E., "Integrating planning and reacting in a heterogeneous architecture for controlling real-world mobile robots", AAAI Conference, 1992.

[5] Musliner D.J., Durfee E.H., Shin K.G., "CIRCA: A Cooperative Intelligent Real-Time Control Architecture", IEEE Trans. on Systems, Man, and Cybernetics, Vol. 23, No. 6, November/December 1993.

[6] Kuhn M.,"A Vision for Linux 2.2 – POSIX.1b Compatibility and Real-Time Support", `ftp://ftp.informatik.uni-erlangen.de /local/cip/mskuhn/misc/linux-posix.1b`

[7] Yodaiken V., Barabanov M., "A Real-Time Linux", Usenix Conference, 1997.

[8] Kansas State's *SLICK WILLIE* Robot, Software, AI Magazine, Vol. 18-1, 1997.

[9] Brooks R.A., Rosenberg C., "L - A Common Lisp for embedded systems", Association of Lisp Users Meeting and Workshop LUV, 1995.

[10] Konolige K., "COLBERT: A Language for Reactive Control in Sapphira", Technical Report, SRI International, June 1997.

[11] Nomadic Technologies, The Nomad 200 User's Manual, 1995.

[12] Real Word Interface, Beesoft 1.2 User's Guide and Reference, 1997.

[13] Simmons R.G., "Structured Control for Autonomous Robots", IEEE Trans. on Robotics and Automation, Vol. 10, No. 1, February 1994.

[14] Tanenbaum A.S., "Modern Operating Systems", Prentice Hall, Englewood Cliffs, NJ.

[15] Lewis B., Berg D.J., "A guide to multithreaded programming - Threads primer", Sunsoft Press, 1996.

[16] The LinuxThreads library web site at `http://pauillac.inria.fr/ xleroy/linuxthreads/`.

[17] Stewart D.B., Khosla P.K., "Mechanisms for Detecting and handling Timing Errors", Communication of the ACM, Vol. 40, No. 1, January 1997.

[18] Stewart D.B., Schmitz D.E., Khosla P.K. , "The Chimera II Real-Time Operating System for Advanced Sensor-Based Control Applications", IEEE Trans. on Systems, Man, and Cybernetics, Vol. 22, No. 6, November/December 1992.

[19] Schoppers M., "A Software Architecture for Hard Real-Time execution of automatically synthesized plans or control laws", Conf. on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS '94).

[20] Zeadally S., "An Evaluation of the Real-Time Performances of SVR4.0 and SVR 4.2", ACM Operating System Review, Vol. 31, No. 1, January 1997.

[21] Liu C.L., Layland J.W., "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", J. of the ACM, Vol. 20, No. 1, January 1973, pp. 46-61.

[22] Beccari G., Caselli S., Zanichelli F., Diemmi D., "Inducing Topological Maps from Task-Oriented Perception and Exploratory Behaviors," 2nd Euromicro Workshop on Advanced Mobile Robots (EUROBOT'97), Brescia, Italy, October 1997.