

# Real Time Control Design for Mobile Robot Fault Tolerant Control.

## Introducing the ARTEMIC Powered Mobile Robot.

Cristian Axenie

Automation and Industrial Informatics Department  
Computer Science Faculty from "Dunarea de Jos" University  
Stiintei Street, Galati, Romania  
Email : cristian.axenie@gmail.com  
Web : http://robotics.viviti.com

Razvan Solea, Lecturer, PhD

Control Systems and Applied Informatics  
Computer Science Faculty from "Dunarea de Jos" University  
Stiintei Street, Galati, Romania  
Email : Daniela.Cernega@ugal.ro

**Abstract** - Real-time applications should timely deliver synchronized data-sets, minimize latency and jitter in their response and meet their performance specifications in the presence of disturbances and faults. The fault tolerant behavior in mobile robots refers to the possibility to autonomously detect and identify faults as well as the capability to continue operating after a fault occurred. This paper introduces a real-time distributed control application with fault tolerance capabilities for differential wheeled mobile robots, named ARTEMIC. The paper focuses on design details and performance analysis at the system operation level. Some stress tests are executed during normal operation to validate the proposal. Specific design, development and implementation details will be provided in this paper.

**Index Terms**—Mobile Robotics, Fault Tolerant Control, Sliding Mode, EKF, Real-Time Linux, Distributed Control

### I. INTRODUCTION

By accepting the challenge to ensure fault tolerant real-time control of a self designed and built wheeled differential mobile robot the author has been able to develop a hierarchical software application that minimizes costs and supports extensibility. Analytical software redundancy is implemented rather than hardware redundancy for fault detection and identification for the minimal robot structure.[4] The main application is designed on a distributed, client-server pattern, based on a TCP/IP wireless communication. The applications base level is built over the interface with the actuators (H-bridge MOSFET driver and 2 DC motors) and sensors (encoders and bumpers) found in the robots structure. The next level is the control and fault tolerance level, to the extent that the control algorithm is implemented here and loops concurrently with the monitoring and fault tolerance task to ensure the robot's autonomy and performance in trajectory tracking operation. The fault tolerant module is based on an set of Extended Kalman Filters used to estimate the current robot position and is using a residual computation and a statistical analysis to determine if a fault occurred in the system and to discriminate between the faults. The third level is a responsible with the communication task, to the extent that it implements a data server and a control server to ensure reliable data and command flows over a wireless network to the other nodes in the distributed system. The client application was designed to meet some specific requirements. The first type of client is responsible of

interacting with the robot operation, basic start, stop, pause actions of the robot, but also with monitoring the status of the robot by receiving specific packets with sensor data. Due to the need of interactivity when injecting the faults a software fault injection framework was developed at this level. Its purpose is to inject a certain fault at a certain moment in time chosen by the client user. ARTEMIC or Autonomous Real Time Embedded Mobile robot Intelligent Controller was developed as the diploma thesis project by the author.

### II. SERVER-SIDE APPLICATION DESCRIPTION

Next an extended description of the server-side application running on the robot embedded computer is given. All architectural and functional aspects will be presented, with focus on both software development elements and implemented control engineering concepts. The server-side application is a real-time Linux C/C++ application. The real-time capabilities were obtained at the OS level by modifying the standard Linux kernel with the RTAI (Real Time Application Interface) patch based on ADEOS (Adaptive Domain Environment for Operating Systems) to the extent that the enhanced micro- kernel can ensure a proper preemption level and I/O latency minimization. The hardware interface with the sensors and actuators is ensured by a data acquisition card (NI-PCI6024E) managed by a powerful kernel-space real-time driver and a rich API in user-space, both part of the COMEDI (Linux Control and MEasurement Device Interface). To properly manage the execution of the two main tasks, namely the control task and the monitoring task the application is organized on an abstract Grafcet (Graphe Fonctionnel de Controle des Etapes et Transitions) which supports the higher level parallelism in the execution of the two tasks. The control task is comprised of two inner speed control PID loops (running at 50ms) for the 2DC motors and a main robot position control loop based on a sliding mode controller (running at 200ms). Concurrently, the monitoring and diagnosis task receives sensor data which is then processed by a set of 5 Extended Kalman Filters every 50 ms. By examining the real and predicted data the application can decide if a fault occurred

**Limited rights client**

Common TCP/IP network Protocol

**Extended rights client**

Common TCP/IP network Protocol

**First abstraction level**  
the C++ application with Linux+RTAI support for specific function calls and device specific driver programming interface as the control server.

**Fault injection framework**

Fault 1	Injection moment
Fault 2	Injection moment
Fault 3	Injection moment
Fault 4	Injection moment

Alter odometry readings by modifying specific parameters according to the fault type

**Robot Control Application**

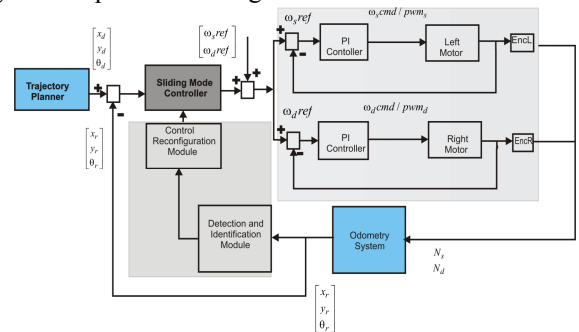
Real Time DAQ card driver specific function Calls

### A. Base level description

To properly understand how the robot control application works, a typical operation context is described, emphasizing class communication during execution. Each sampling period the application reads the data through the DAQ card from the encoders and

### B. Control algorithm level description

As mentioned earlier the robot is operating in trajectory tracking mode so its main objective is to follow a certain trajectory under time constraints. The control algorithm goal is to minimize the position errors, namely the longitudinal error, the lateral error and the heading error. To ensure proper error convergence in the presence of disturbances and modeling uncertainties a sliding mode controller was chosen. A sliding mode controller based on the kinematic model of the robot was used. By using a sliding mode controller a trade-off between tracking performance and parametric uncertainty was made, to the extent that it provides a good solution in maintaining the stability and consistent performance in the face of modeling imprecision and disturbances.[2] Next the complete robot control system is presented in figure3.



The robot should track the reference parametric curve with time constraints and real-time control signal computation is critical. The idea behind the trajectory tracking operation mode considers a virtual robot that tracks the reference trajectory with no error and "pulls"

the real robot after it to the extent that the controller reacts to minimize the errors. Let the robot be defined by the next position vector (state vector)  $[x_r, y_r, \theta_r]^T$  and we define the error vector as  $[x_e, y_e, \theta_e]^T = [x_d, y_d, \theta_d]^T - [x_r, y_r, \theta_r]^T$ , (1) where  $[x_d, y_d, \theta_d]^T$  is the virtual robot state vector, the set-point vector to track. Next the robot's kinematic model and tracking error computation is considered.

$$\begin{cases} \dot{x}_r = v_r \cos \theta_r \\ \dot{y}_r = v_r \sin \theta_r \\ \dot{\theta}_r = \omega_r \end{cases}, \begin{cases} \dot{x}_d = v_d \cos \theta_d \\ \dot{y}_d = v_d \sin \theta_d \\ \dot{\theta}_d = \omega_d \end{cases}, \begin{bmatrix} x_e \\ y_e \\ \theta_e \end{bmatrix} = \begin{bmatrix} \cos \theta_d & \sin \theta_d & 0 \\ -\sin \theta_d & \cos \theta_d & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_r - x_d \\ y_r - y_d \\ \theta_r - \theta_d \end{bmatrix}, \quad (2)$$

The main objective was to design a stable and robust controller for robot position control that should output a control signal vector  $[v_c, \omega_c]$  (linear and angular speeds) in the presence of disturbances. The error derivative is given by

$$\begin{aligned} \dot{x}_e &= -v_d + v_r \cos(\theta_e) + y_e \omega_d \\ \dot{y}_e &= v_r \sin(\theta_e) - x_e \omega_d \\ \dot{\theta}_e &= \omega_r - \omega_d \end{aligned}, \quad (3)$$

where the set-point values for the speed control loops are  $v_d$  and  $\omega_d$ . [8] The set-point for the main robot position control loop is generated by a trajectory planner based on 5th degree equations for smoothing the robot's operation and control effort. The trajectory planner outputs linear and angular speed and acceleration set-points at each sampling period. These values are transformed using inverse kinematics equations in vector form  $[x_d, y_d, \theta_d]^T$ . The next step was to design the switching surfaces for the sliding mode controller such that it can ensure proper error convergence. The first switching surface was chosen to ensure lateral error convergence and is given by the next equation  $s_1 = \dot{x}_e + k_1 x_e$ , (4) To ensure proper longitudinal and heading error convergence the second switching surface combines the two objectives and is given by  $s_2 = \dot{y}_e + k_2 y_e + k_0 \text{sgn}(y_e) \theta_e$ , (5), where the  $k_1, k_2, k_3$  parameters are specific weights associated to each error component. Practically the control signal could be computed using  $s' = -Qs - P \text{sgn}(s)$ , (6), where  $Q, P$  are positive constant values that are determining the error convergence speed. [5] By deriving the surfaces equations and replacing the error components one can easily obtain the control signal vector under the form of

$$v_c = \frac{-Q_1 s_1 - P_1 \text{sgn}(s_1) - k_1(x_e) - \omega_d y_e - \omega_d y_e + v_r \dot{\theta}_e \sin(\theta_e) + v_d}{\cos(\theta_e)}, \quad (7)$$

$$\omega_c = \frac{-Q_2 s_2 - P_2 \text{sgn}(s_2) - k_2(y_e) + \omega_d x_e + \omega_d x_e - v_r \sin(\theta_e)}{v_r \cos(\theta_e) + k_0 \text{sgn}(y_e)} + \omega_d$$

To ensure a good convergence speed of the robot state

vector to the switching surface (which in fact gives the desired system dynamics) and also to eliminate chattering (caused by fast control signal switches) the  $P$  and  $Q$  terms should be cautiously chosen. To ensure the stability of the system we defined a Lyapunov candidate function given by  $V = 0.5 s^T s$  and with  $\dot{V} = s_1 \dot{s}_2 + s_2 \dot{s}_1 = -s^T Qs - P_1 |s_1| - P_2 |s_2|$ , (8), the main condition is that  $V \dot{V} < 0$  achieved if  $Q_i \geq 0$  and  $P_i \geq 0$ . [2] As mentioned earlier the sliding mode controller outputs angular speed set-points for the inner PID loops. The PID controllers for the two DC motors were implemented using first-order-hold method. The PID loops are running at 20Hz and the specific parameters were tuned to meet a fast response time (0.5s) and a small overshoot (10%). [10] Next some results from the closed loop fault-free operation of the robot are given to analyze the robot behavior on a specific trajectory.

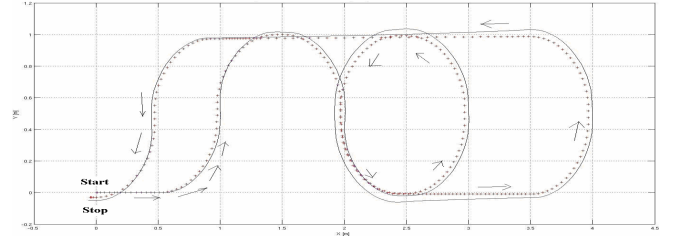


Fig. 4. Robot operation in trajectory tracking with sliding mode controller in fault free operation.

The real robot trajectory is represented by a continuous line and the reference trajectory is given with crosses. The movement direction is indicated by arrows. As one can see the position error is kept in acceptable limits by the robust sliding mode controller and the robot can achieve its objective in time. Once more one can observe that good performance is achieved in using the sliding mode controller. Another important aspect regards the analysis of the error components in the fault free operation, because it can be considered as a proper validation for the controller. [6] Next the three error components are depicted.

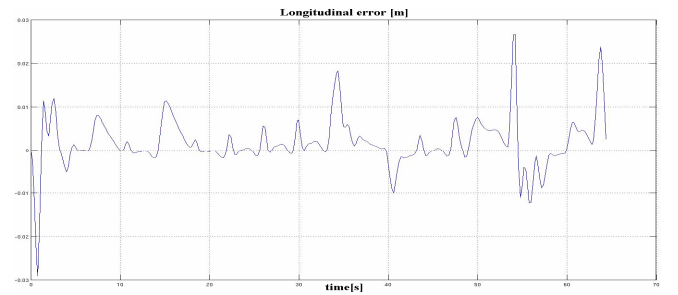


Fig. 5. Robot longitudinal error  $x_e$ .

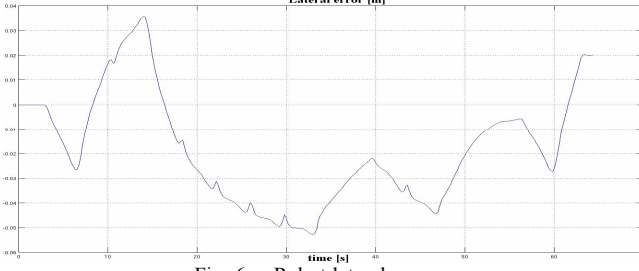


Fig. 6. Robot lateral error  $y_e$ .

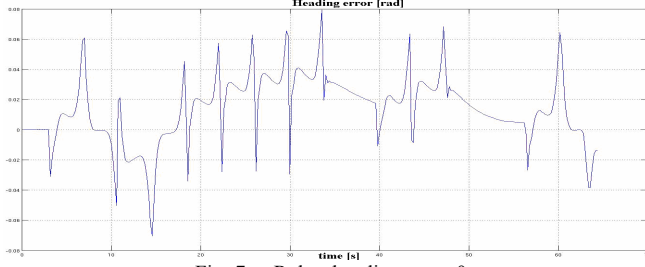


Fig. 7. Robot heading error  $\theta_e$ .

One can see that the error convergence is properly ensured by the controller. The longitudinal error in the fault free operation is under 3cm, the lateral error is under 6cm and the heading error is under 8rad. Experiments have shown that even for simpler trajectories (simple 2m line, 0.75m radius circle) the controller behaves well and despite the fact that the robot structure had some hardware limitations (encoder resolution, 500PPR) the real-time control goals were met even for slightly high sampling periods; all the results are validated with the earlier error analysis. Following the next level of the application is presented.

### C. Monitoring and diagnosis level description

Due to hardware limitations in injecting faults the application is also responsible to inject faults by software using a fault injection framework. By benefiting of a certain degree of robustness given by the sliding mode controller the application resides on a simple mechanism for fault detection, identification and support for control reconfiguration based on the Extended Kalman Filter (EKF). So, the application implements a set of 5 EKF that form the fault detection and identification module. The current application considers a fault benchmark comprised of faults manifested by system's parameter variation specifically mechanical faults that alter the robot behavior. Hence two fault classes were defined each containing two types of faults. The first class of faults introduces the flat tire fault for each of the robot wheels. At the implementation level this fault exhibits by diminishing the wheel radius with a certain value. As a consequence the whole kinematic model of the robot is modified and visible modifications in the robot's behavior will be present. The second class of faults

introduces a periodic variation of the wheel radius (a bump, caused hypothetically by an object attached to the wheel). Hence, for a small time period the variation will manifest itself and alter the robot operation. As mentioned earlier a fault injection framework was implemented and its main purpose is to alter the values from the sensors in odometry specific computations. Hence, when no fault is injected the robot will behave normally (as presented when the control level was described) but when a fault was injected the robot position determining system (in fact the control system feedback) will feed altered data to the controller as if a physical fault occurred. Next a depiction of the fault detection and identification module is given.

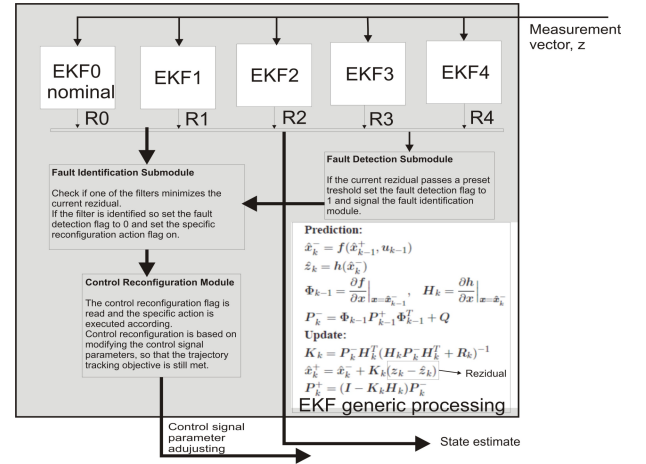


Fig. 8. Fault detection and identification module.

Each of the five EKF encapsules a kinematic model of the robot but with different parameters. The main idea resides on the fact that for the same input vector,  $z$ , with noise, each filter generates a prediction of the robot's state vector. Each filter is associated with a certain fault type. EKF0 is the nominal filter corresponding to the fault free robot operation. EKF1 filter contains the same robot kinematic model but with modified parameters to emphasize the right tire flat fault, so the right wheel radius has a smaller value. Hence, the prediction of EKF1 will be the robot state vector if a right tire flat occurred. The EKF2 filter is similar to EKF1 excepting it encapsules the left tire flat fault specific model. In the same way the other two filters associated with the second fault class, EKF3 and EKF4, are predicting the robot's state vector in the presence of a second fault class fault. Besides the state estimate each EKF generates a measurement vector estimate during the prediction stage, which is used in the correction stage of the filter.[3] Basically the detection method is based on a nominal residual computation and a simple thresholding test on the nominal residual which can output that a fault occurred. It is the only one affected by the fault occurrence so it will step out from some preset value



intervals. When identifying the fault the application just finds the smallest residual from the 5 ones, because when a fault occurs the main measurement vector is altered and it nears the predicted measurement vector by the EKF encapsulating that fault behavior.[1] Next, specific implementation details of the EKFs are presented. Starting from the generic structure during the implementation some simplifying hypothesis were issued and we know that  $Q$  is the process noise covariance matrix,  $R$  is the measurement covariance matrix,  $\sigma_x = \sigma_y = \sigma_\theta = 0.1$  are the standard deviations for the process noise,  $\sigma_{\text{meas}}=0.01$  is the standard deviation of the measurement noise. To correctly detect if a fault occurred some statistical tests on the residual are required.[9] The statistical parameters of the residual (mean and variance) will be compared on-line at each sampling period with the values obtained for these parameters in fault-free operation. In fault-free operation the residual is a white noise sequence with 0 mean and with a variance of

$\eta_k = H_k(\hat{x}_k^-)P_k^-H_k^T(\hat{x}_k^-) + R, (9)$ . Basically if a fault occurs in the system it will determine a modification of the next residual based standard sequence,

$$\eta_s = (H_k(\hat{x}_k^-)P_k^-H_k^T(\hat{x}_k^-) + R)^{-1/2}(y_k - h_k(\hat{x}_k^-)) = \eta_k^{-1/2}r_k, (10)$$

The goal is to determine the estimate of the real value of a sample given using the next relation where  $N$  is the total samples number,

$$\hat{\eta}_s = \frac{1}{N} \sum \eta_{sj}, (11)$$

So, in the hypothesis  $H_0$  of no fault  $\hat{\eta}_s$  has a gaussian distribution with 0 mean and  $1/N$  covariance. Over a certain acceptance threshold the  $H_0$  hypothesis is no longer true and so a new hypothesis  $H_1$  is now true marking a fault occurrence. Next some experimental results are presented. First it is useful to analyze the residuals generated by the nominal filter when a fault occurs, in fact these residuals are those predicted by the fault embedding filters, from the moment the fault occurs. For example if a right tire flat fault occurs at sample 100 in the execution the residual evolution from 100th sample is the predicted residual by the EKF1.

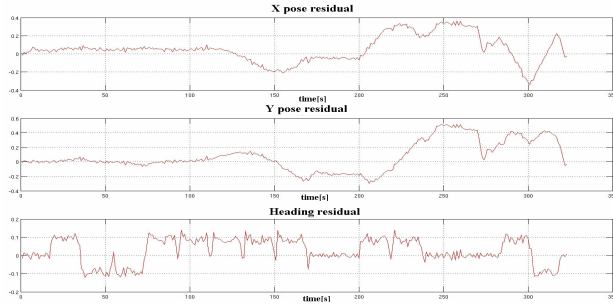


Fig. 9. Residual analysis for first fault class.

From this residual analysis one can see that each EKF is sensitive to a certain fault type and using these properties

the detection and identification modules were built. Assuming that a right tire flat fault was injected at moment  $t=10s$  and the new radius of the wheel is 3cm smaller than the initial value the behavior of the robot is altered as one can see in the next figure(\* ref, - robot).

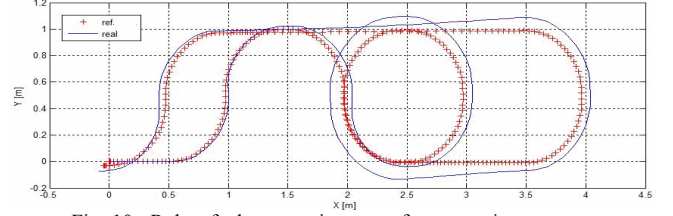


Fig. 10. Robot faulty operation vs. reference trajectory.

Next an extended position error analysis over the fault free and faulty operation is given (red marks fault free operation and blue marks the faulty operation).

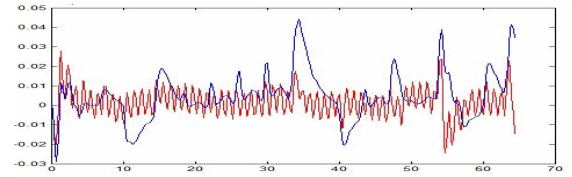


Fig. 11. Longitudinal error comparison fault-free/faulty operation.

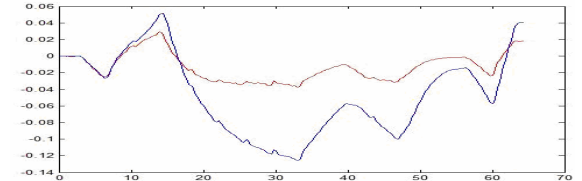


Fig. 12. Lateral error comparison fault-free/faulty operation.

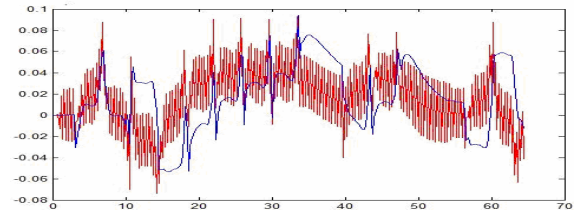


Fig. 13. Heading error comparison fault-free/faulty operation.

As one can see there are major variations from the nominal behavior after a fault occurred in the system. It is important to mention that the thresholding mechanism reacts if at least two components of the residual vector are bigger than the preset values. So, the fault could be visible in all the residual components or not. As a typical operation context when a fault is detected, it is then identified and the client application is informed what kind of fault occurred. At the moment only one fault can be detected and identified, no fault queue or priority system are implemented. Following a brief description of the client application is given.

### III. THE CLIENT-SIDE APPLICATION DESCRIPTION

The two main components of the application are linked through a wireless communication ensuring the client application to be responsible with the external events that determine the robot operation. It implements a similar structure with the embedded application and supports the fault injection framework. Next a synthetic image of the Java GUI client application is given.

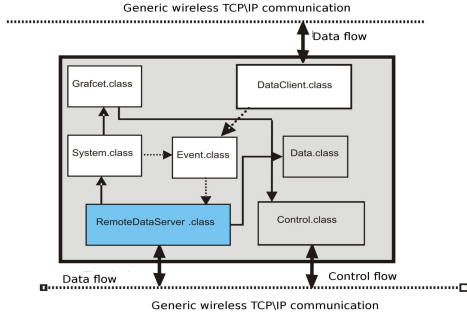


Fig. 14. Client-side Java application architecture.

Besides the interactive task of injecting faults the client application is responsible to choose the Grafcet structure that encapsules the execution rules for the control algorithm and the specific supported actions (start/pause/stop) for the robot operation, also it offers the possibility to choose between multiple trajectories and to visualize the robot's current position. Using a full load of the embedded system latency measurements were taken. So, by using a 50ms sampling period for the inner control loop and the monitoring and diagnosis tasks and a 200ms sampling period for the robot position controller good average ms level latencies were obtained.

The two possible Grafcet execution schemes serial/ parallel assume a serial or parallel execution of user defined tasks for the robot. Currently the two implemented tasks, real-time control and monitoring and diagnosis must run concurrently to ensure consistent and valid results in the fault tolerant control problem.

Using a full load of the embedded system latency measurements were taken. So, by using a 50ms sampling period for the inner control loop and the monitoring and diagnosis tasks and a 200ms sampling period for the robot position controller good average ms level latencies were obtained. Some measured latencies for the standard, Low-Latency, preemptible and preemptible with lock-breaking kernels while running different loads in background. All the experiments were performed using the Latency Benchmark tool [7].

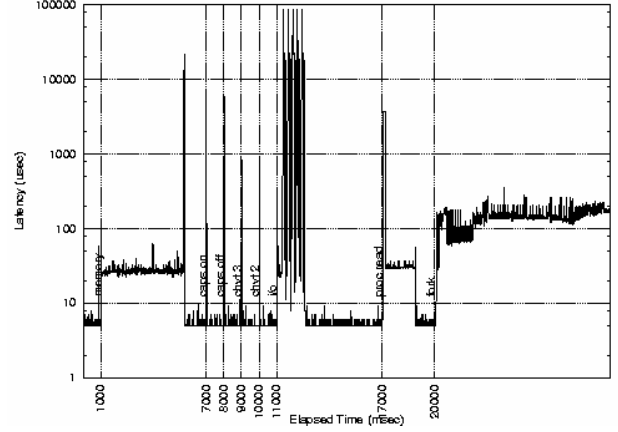


Fig. 17. Latency in the Standard Kernel.

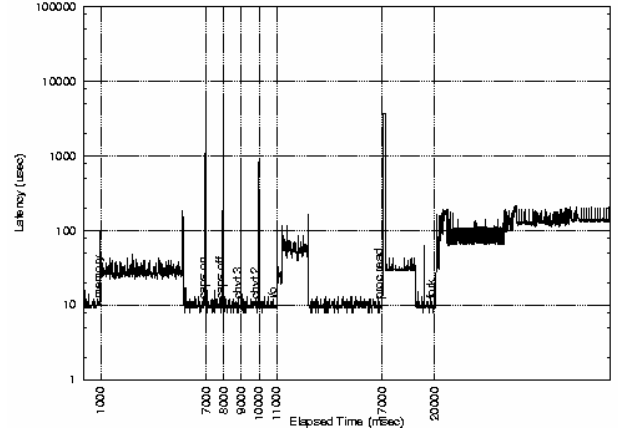


Fig. 18. Latency in the Low Latency Kernel.

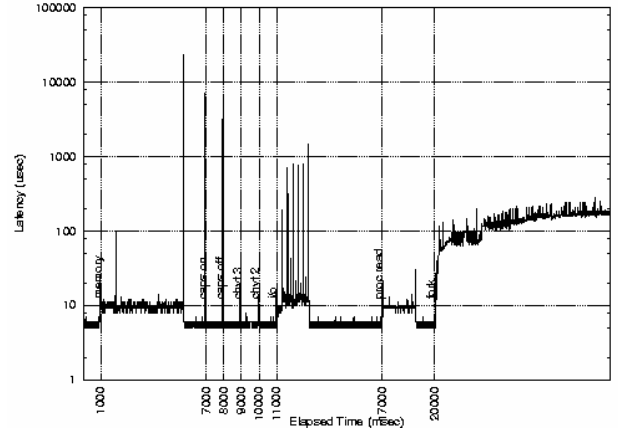


Fig. 19. Latency in the preemptible Kernel.

