REAL TIME LINUX: TESTING AND EVALUATION

Phil Wilshire
Lineo, Education Services
email address philw@lineo.com

Abstract

This paper discusses the different benchmarking tools used to evaluate the performance of Linux and their suitablilty for evaluating Real Time system Performance.

The author will then present an Open Source Test and Evaluation Suite toolkit which rapidly allows system performance to be monitored and calibrated in a manner to allow a user to define the relative merits of each proposed system.

1 Introduction

Linux is capable of truly exceptional performance on even modest platforms. It can respond to an interrupt in a few microseconds under certain, ideal circumstances.
This performance can quickly disappear when the load on the test system is increased. The process of sharing the cpu between different tasks causes a gradual but, significant, performance degradation. Real Time Linux provides a separate, preemptable, run time environment that does not suffer from the same performance losses.
Testing for Real Time performance can very quickly become complicated and heated discussions are soon started. The introduction of external measurement tools introduces another level of system disturbance due to the Input / Output Performance of typical x86 based systems.
The call for a simple, effective test method that can test the software performance and capability is often heard.
This document will describe a simple test suite that can obtain results from any pentium based system.
The actual test performed is somewhat basic but the suite can be extended to use more complex tests as required.

2 Established Benchmarks and problems

There are a number of well know established benchmarks [1] used to evaluate Linux and other operating systems. These concentrate on different features of the operating system and are used to focus on particular areas of performance. Larry McVoy's lmbench is probably the most respected performance measurement tool published today.
see http://www.bitmover.com/lmbench/
Other examples can be found at this site:-
http://www.kernelbench.org/links.html
When testing lmbench I found a significant variation in the results obtained from the same supposedly "unloaded" system. The variation in the test results sometimes exceeded the actual value being monitored.
However, tools are excellent when used determining the raw capablilty of a system they tend to be less useful for Real Time performance measurement.
The speed of a context switch or of memory access provides important data but, with a Real Time system, there are other equally important measurements.

3 Real Real Time ?

Real Time systems require many of the same performance characteristics as Non Real Time Systems. There is, in addition, one special requirement unique to Real Time. The requirement to absolutely meet a hard deadline under a wide range of use cases.
It could be argued that all the other Operating systems characteristics can be reduced to this simple problem.
The task switching and interrupt response times can be usefully represented by stating the system's ability to meet a scheduled deadline.

Such a deadline need not be executed very quickly as long as the execution time and any variation or jitter can be bounded and defined.
This characteristic is called "Periodic Scheduling Accuracy" for the purposes of this paper.

3.1 Periodic Scheduling Accuracy

A simple test task is run which collects a number of data samples. The task measures the time at which it starts to run.
The system is placed under different loads for different test runs. The results are tabulated.
The three load conditions selected are :-

- No load - The system is idle with cron turned off and no other tasks running
- Ping - The system pings another node in the network.
- Disk - the system continually writes a disk file

3.2 Ping Code

The background task for the Ping test.

```
#!/bin/sh
#
# $Id: run.ping,v 1.2
# 2000/08/29 08:26:17 gss Exp $
#
# shell script to exercise the network system

if [ ! -d "$1" ]; then
  echo run.ping needs a directory parameter
  exit 1
fi

if [ -z $2 ]
then
  PNODE="192.168.2.101";
else
  PNODE=$2
fi

# sh run.ping
./savepid ./run.ping $1

while (true) do
  ping -f -c 10000 $PNODE #> /dev/null
done
```

3.3 Disk Code

The background task for the Disk Access test.

```
#!/bin/sh
# $Id: run.disk,v 1.1
# 2000/08/28 23:56:18 gss Exp $
# shell script to exercise the disk system
# sh run.disk

if [ ! -d "$1" ]; then
  echo run.disk needs a directory parameter
  exit 1
fi

./savepid ./run.disk $1

while (true)  do
  dd if=/dev/zero of=/tmp/disk.dummy bs=1024 \
    count=4096  #> /dev/null
  sync
  sync
done
```

4 Time Measurement

Having decided that some form of measurement of periodic scheduling accuracy is a suitable metric to apply to real time systems we are left with the problem of measuring the time that the task actually started its execution.
In previous experiments external equipment has been attached to a system and pulses produced on the parallel port were used to provide timing data.

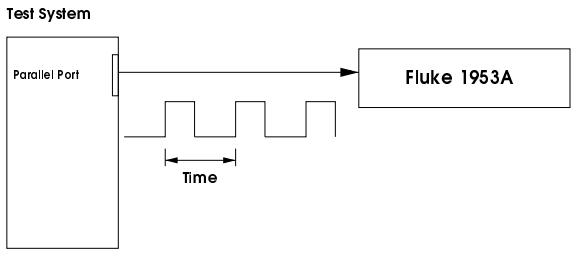

FIGURE 1: *An Example Test System*

Work by John Storrs [2] and others has indicated that significant delays can be introduced by the PC I/O architecture to severely disrupt this mechanism. A I/O delay of 10's of Microseconds can severely disturb a measurement of under 10 uSeconds.
A system originally used by
Universidad Nacional de La Plata, Argintina
http://www.fisica.unlp.edu.ar/
provides some useful data.
But since this uses an output device on the I/O bus the measurement will be subjected to I/O delays.
The Pentium Processor comes to our help.
This device contains a program cycle counter that can be readily accessed by software.

```
/* Inline function to return CPU cycle count */

static inline long long rtrdtsc(void)
{
  long long time;
  __asm__ __volatile__( "rdtsc" : "=A" (time));
  return time;
}
```

All that remains is the measurement of the actual cpu clock to translate these cycle counts into a real time measurement. The following code will provide a good measurement of the actual Cpu Clock.

4.1 Clock Calibration Code

This is the code used to calibrate the clock. It runs together with the RTAI Real Time Linux Extensions. Similar code could be prepared to run under RTL.

```
/* This code requires a version of RTAI to be loaded to give access to
 Real Time timers */
#define MODULE
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/signal.h>
#include <linux/sched.h>
#include <asm/io.h>

#include <rtai.h>

const int SECS = 5;

MODULE_PARM(SECS, "i");

static int RESET_COUNT;

static void calibrate(void)
{
  static int count = 0, gcount = -1;
  static RTIME tbase;
  long linux_cr0, fpu_reg[27];
  double freq;
  union {unsigned long long time;
         unsigned long time_lh[2]; } tsc;
  tsc.time =  rd_CPU_ts();
  if (gcount < 0) {
      tbase = tsc.time;
  }
  gcount++;
  if (++count == RESET_COUNT) {
    tsc.time -= tbase;
    __asm__ __volatile__("movl %%cr0,%%eax": "=a" (linux_cr0): : "ax");
    __asm__ __volatile__("clts; fnsave %0": "=m" (fpu_reg));
    freq =  (double)tsc.time_lh[1]*(double)0x100000000LL + (double)tsc.time_lh[0];
    count =  (freq*CLOCK_TICK_RATE)/(((double)gcount)*LATCH) + 0.4999999999999;
    __asm__ __volatile__("frstor %0" : "=m" (fpu_reg));
    __asm__ __volatile__("movl %%eax,%%cr0": :"a" (linux_cr0): "ax");
    printk("\n->>> MEASURED CPU_FREQ: %d (Hz) (%d s) <<<-\n", count, gcount/100 + 1);
    count = 0;
  }
  rt_pend_linux_irq(0);
}

int init_module(void)
{
```

```
    RESET_COUNT = SECS*100;
    rt_mount_rtai();
    rt_request_global_irq(TIMER_8254_IRQ, calibrate);
    printk("\n->>> HERE WE GO (PRINTING EVERY %d SECONDS) <<<-\n\n", RESET_COUNT/100);
    return 0;
}

void cleanup_module(void)
{
    rt_free_timer();
    rt_free_global_irq(TIMER_8254_IRQ);
    rt_umount_rtai();
}
```

5 Test Method

There are two types of simple measurement tasks used. One is for Real Time and the other is for Non Real Time applications.

5.1 Non Real Time

The System runs a simple task that will sleep for a given period of time. When the task wakes up it will take a measurement of the CPU cycle count. The time between this and the last wake up period is then saved in a memory array. When the test is over the data in the array is printed to a file.
The code for the non real time task is given.

```
////////////////////////////////////////////////////////////////////////
//
// $Id: pthsqlsamp.c,v 1.2 2000/08/29 19:37:28 gss Exp $
//
//
// Uses code from:-
//       Andris Pavenis <pavenis@lanet.lv>
//         to get the timer data
//
//  Heavily based Original Work from :-
//      Departamento de Fsica
//      Universidad Nacional de La Plata
//
//  compile with the -O2 optimisation
//
////////////////////////////////////////////////////////////////////////
// sqlsamp.c used to measure processor sched accuracy uses the cpu clock.
// Copyright (C) 2000 Zentropic Computing, Inc.
//
// This program is free software; you can redistribute it and/or
// modify it under the terms of the GNU General Public License
// as published by the Free Software Foundation; either version 2
// of the License, or (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
//
// AUTHOR: Phil Wilshire <philw@lineo.com>
// DATE  : Mon Jun 6 2000
```

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <time.h>

#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sched.h>


#define DEF_PERIOD 50   /* every 50 milli Seconds  */
#define DEF_COUNT 50   /* 50 samples  */
#define DEF_RESTYPE "psaidle";

// Inline function to return the CPU clock counter
//
static inline long long lrdtsc(void)
{
  long long time;

  __asm__ __volatile__( "rdtsc" : "=A" (time));

  return time;
}


static void usage (char * prog ) {
  printf(" useage :- \n");
  printf(" %s cpu_freq [num_samples] [cycle_delay]\n",prog);
  return;
}


int main( int argc , char * argv[]) {

  struct timespec ts;
  struct timespec tsr;
  long count, mcount, mperiod, value;

  unsigned long long t1;
  unsigned long long t2;

  double * tdr;
  double cpu_freq;

  char * restype;

  struct sched_param mysched;

  mysched.sched_priority = sched_get_priority_max(SCHED_FIFO) -1;
  if ( sched_setscheduler (0,SCHED_FIFO, &mysched ) == -1 ) {
    printf("Error in steting the scheduler \n ");
    perror("errno");
    exit(0);
  }

  // There are two required parameters
  if ( argc < 2 ) {
```

```
  usage (argv[0]);
  exit(0);
}

// The CPU frequency in MHz is a required parameter
sscanf(argv[1], "%lf",&cpu_freq);
printf("#cpu freq = %f \n",cpu_freq);

// The sample count is a required parameter
sscanf(argv[2], "%ld",&mcount);
printf("#mcount = %ld \n",mcount);

// period in mSecs -- convert to usecs
mperiod = DEF_PERIOD;
if ( argc > 3 ) {
  sscanf(argv[3], "%ld",&mperiod);
  printf("#period = %ld \n",mperiod);
}
mperiod *= 1000;

// restype IDLE, DISK or PING
restype = DEF_RESTYPE;
if ( argc > 4 ) {
  restype = argv[4];
  printf("#restype = %s \n",restype);
}

printf("#\n");

mlockall ( MCL_CURRENT | MCL_FUTURE );
tdr = (double *) malloc(mcount * sizeof(double));
if (tdr == NULL ) {
  exit(0);
}

for( count=0 ; count<=mcount ; count++ ) {

  ts.tv_sec = 0;
  ts.tv_nsec = mperiod * 1000;
  tsr.tv_sec = 0;
  tsr.tv_nsec = 0;

  t1 = lrdtsc();
  nanosleep(&ts,&tsr);
  t2 = lrdtsc();

  // The elasped cycles over the frequency in MHz is usecs.
  tdr[count] = ((t2-t1) / cpu_freq);
}

// Don't use the first measurement
// Note that 10ms--the standard Linux lag on nanosleep
// have been factored out of the error time
for( count=1 ; count<=mcount ; count++ ) {
  value = (long)(tdr[count] + (double)0.5);

  printf("insert into data (results_id,data,rawdata) ");
  printf("VALUES (%sRES_ID,%ld,%ld);\n", restype, value-mperiod-10000, value);
}
```

```
  free((char *) tdr);
  return 0;
}


/*
* Local variables:
*  compile-command: "gcc  -Wall -Wstrict-prototypes -O2 -o pthsqlsamp pthsqlsamp.c "
*  c-indent-level: 4
*  c-basic-offset: 4
*  tab-width: 4
* End:
*/
```

5.2 Real Time

A task is run under a real time scheduler to measure the Processor Cycle count. This task delivers the data to a user task for saving in the memory array. When the test completes the data is printed to a file.

The Real Time code follows:-

```
//////////////////////////////////////////////////////////////////////////////
// $Id: rttask.c,v 1.2 2000/09/14 19:38:57 gss Exp $
//  rttask.c
//
//////////////////////////////////////////////////////////////////////////////
//
// Copyright  2000 Zentropic Computing LLC.
//
// Permission is hereby granted, free of charge, to any person obtaining a
// copy of this software and associated documentation files (the "Software"),
// to deal in the Software without restriction, including without limitation
// the rights to use, copy, modify, merge, publish, distribute, sublicense,
// and/or sell copies of the Software, and to permit persons to whom the
// Software is furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in
// all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.  IN NO EVENT SHALL
// ZENTROPIC COMPUTING LLC BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
// FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
// DEALINGS IN THE SOFTWARE.
//
// Except as contained in this notice, the name of the Zentropic Computing LLC
// shall not be used in advertising or otherwise to promote the sale, use or
// other dealings in this Software without prior written authorization from the
// Zentropic Computing LLC
//
// Authors:         Phil Wilshire
// Original date:       Wed May 31, 2000
// Id:          @(#)$Id: rttask.c,v 1.2 2000/09/14 19:38:57 gss Exp $
//
// Description:     This code collects data for the histogram
//              application
//
//////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////
```

```
// Data and declarations common to user/kernel
////////////////////////////////////////////////////////////////////////////


#define HIST_FIFO       0

////////////////////////////////////////////////////////////////////////////
//
// kernel module code
//
////////////////////////////////////////////////////////////////////////////
#include <linux/module.h>
#include <asm/io.h>
#include <rtai.h>
#include <rtai_sched.h>
#include <rtai_fifos.h>
#include <math.h>


int PERIOD;    // task period in microseconds

MODULE_PARM(PERIOD,"i");

#define T_FREQ 5    // task frequency Hz


int     task_ticks = 0;

RT_TASK task_str;

// Inline function to return CPU cycle count
//
static inline long long rtrdtsc(void)
{
  long long time;

  __asm__ __volatile__( "rdtsc" : "=A" (time));

  return time;
}

// The real-time task which puts the current CPU clock count
// into the FIFO, then goes back to sleep
//
static void periodic_func(int arg)
{
  long long this_time;

  while( 1 )
  {
    rt_task_wait_period();
    this_time  = rtrdtsc();
    rtf_put( HIST_FIFO, &this_time, sizeof(long long) );
  }
}

// Kernel task init.  Create the FIFO to use plus a task
// (periodic_func()) that runs based on PERIOD.
//
int init_module(void)
```

```
{
  int err;
  RTIME small_delay;

  if( PERIOD <= 0 )
    PERIOD = 200000;

  rt_set_oneshot_mode();
  task_ticks = nano2count(PERIOD * 1000);

  small_delay = nano2count(100000);

  printk(" task_ticks = %d \n",task_ticks);
  start_rt_timer(task_ticks);
  rt_linux_use_fpu(1);

  // initialise the fifo used to wake up the master user timer
  if((err = rtf_create(HIST_FIFO, 5000)) < 0)
  { printk("rtf_create: rtfifo %d :errno = %d\n", HIST_FIFO, err);
    return -1;
  }
  printk(" warning fixed period here \n");
  rt_task_init(&task_str, periodic_func, 0, 3000, 4, 1, 0);
  rt_task_make_periodic(&task_str, rt_get_time()+small_delay, task_ticks);

  return 0;
}

// Kernel task cleanup.
//
void cleanup_module(void)
{
  stop_rt_timer();
  rt_task_suspend(&task_str);
  rt_task_delete(&task_str);
  rtf_destroy(HIST_FIFO);

}

/*
* Local variables:
*  compile-command: "gcc  -DMODULE -D__KERNEL__ -I /usr/src/rtai/include \
    -Wall -Wstrict-prototypes -O2 -c -o rttask.o rttask.c"
*  c-indent-level: 4
*  c-basic-offset: 4
*  tab-width: 4
* End:
*/
```

Some user space code to record the execution times measured by the real time task.

```
////////////////////////////////////////////////////////////////////////
//
// rtdata.c
//    User space task used to receive the realtime data from the fifo
//
////////////////////////////////////////////////////////////////////////
//
// Copyright (C) 2000 Zentropic Computing, Inc.
```

```
//
// This program is free software; you can redistribute it and/or
// modify it under the terms of the GNU General Public License
// as published by the Free Software Foundation; either version 2
// of the License, or (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
//
// AUTHOR: Phil Wilshire <philw@lineo.com>
// DATE  : Wed Jun 7 2000

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include <rtai_fifos.h>

#define DEF_PERIOD 50   /* every 50 milli Seconds  */
#define DEF_COUNT 50   /* 50 samples  */
#define DEF_RESTYPE "psaidle";


static void usage (char * prog ) {
  printf(" useage :- \n");
  printf(" %s cpu_freq [num_samples] [cycle_delay]\n",prog);
  return;
}


int main(int argc , char * argv[])
{
  int fd0;
  long long lsamp;

  double cpu_freq;
  long count, mcount, mperiod, diff;
  double * tdr;
  double tdf;

  char * restype;

  struct sample { long long idata; } samp;

  // At least 2 parameters are required
  if ( argc < 2 ) {
    usage (argv[0]);
    exit(0);
  }
```

```
// CPU frequency is a required parameter
sscanf(argv[1], "%lf",&cpu_freq);
printf("#cpu freq = %f \n",cpu_freq);

// Test run count is a required parameter
sscanf(argv[2], "%ld",&mcount);
printf("#mcount = %ld \n",mcount);

// Task frequency in msecs -- convert to usecs
mperiod = DEF_PERIOD;
if ( argc > 3 ) {
  sscanf(argv[3], "%ld",&mperiod);
  printf("#period = %ld \n",mperiod);
}
mperiod *= 1000;

// restype IDLE, DISK or PING
restype = DEF_RESTYPE;
if ( argc > 4 ) {
  restype = argv[4];
  printf("#restype = %s \n",restype);
}

// Allocate space for data
tdr = (double *) malloc(mcount * sizeof(double));

if (tdr == NULL ) {
  exit(0);
}

// Open the FIFO to read
if ((fd0 = open("/dev/rtf0", O_RDONLY)) < 0) {
  fprintf(stderr, "Error opening /dev/rtf0\n");
  exit(1);
}

// Read the first few value so we can start getting differences
for( count=0 ; count<5 ; count++ ) {
  read(fd0, &samp, sizeof(samp));
  fflush(stdout);
}
lsamp = samp.idata;

for( count=0 ; count<mcount ; count++ ) {
  // Read a sample, calculate and save difference from previous sample
  read(fd0, &samp, sizeof(samp));

  tdf = (double) ((samp.idata-lsamp) / cpu_freq);
  lsamp = samp.idata;
  tdr[count] = tdf;

  fflush(stdout);
}

printf("# cpu_freq = %f MHz count = %ld per = %ld usec\n", cpu_freq, mcount, mperiod);

// Print rounded differences as error and raw value
for ( count=0 ; count<mcount ; count++ ) {
  diff = (long)( tdr[count] + (double)0.5);
```

```
    printf("insert into data (results_id,data,rawdata) ");
    printf("VALUES (%sRES_ID,%ld,%ld);\n", restype, diff-mperiod, diff);
  }

  free((char *) tdr);

  return 0;
}

/*
* Local variables:
*  compile-command: "gcc  -Wall -Wstrict-prototypes -I/usr/src/rtai/include \
    -O2 -o rtdata rtdata.c "
*  c-indent-level: 4
*  c-basic-offset: 4
*  tab-width: 4
* End:
*/
```

Target Software	Category	Load	Worst Case Accuracy (1s 100ms 10ms 1ms 100us 10us 0)
RTAI 1.3	Hard Realtime	Disk Load	17us
RTL 2.3	Hard Realtime	Disk Load	19us
Red Hat 6.2 Low Latency	Low-Latency	Disk Load	4053us
Red Hat 6.2	SCHED_FIFO	Disk Load	9706us
MontaVista 1.0	Low-Latency	Disk Load	49065us

FIGURE 2: *Example Test Results*

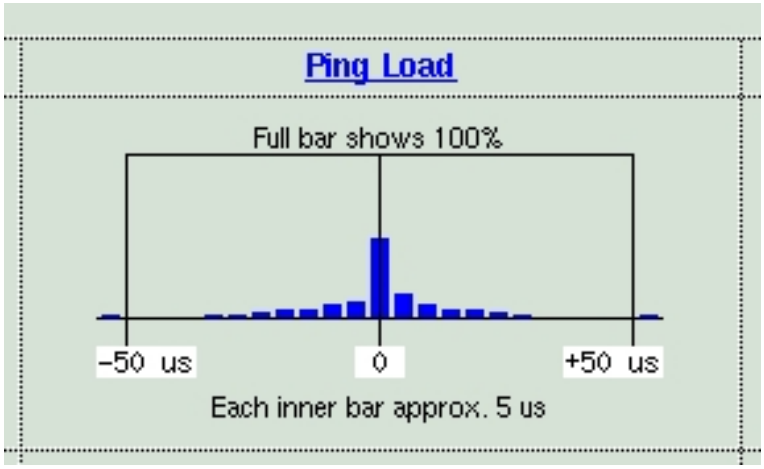

FIGURE 3: *Example Test Histogram*

5.3 Post processing.

The output from the code is "almost" an SQL file. This, after some substitutions, is suitable for placing into a database.

The timing data is then stored in a database and extracted and plotted on a web page.

6 Statistics

When running a test such as this some means of predicting the probability of extending the results obtained under test conditions to the conditions present in the Real World is required.

A histogram of the results provides a useful indication of the quality of the test results.

A 1 sigma measurement will determine a range within which 68% of all samples are predicted to fall within (for a true Normal Distribution). You get 99.99% for a 4 sigma.
More details on the use of Statistics can be found here.
http://biology.nebrwesleyan.edu/
empiricist/sources/tips/stats1.html
and here
http://www.statsoftinc.com/
textbook/esc.html
(Why the "Normal distribution" is important)
When designing a Real Time System the 3 or 4 sigma figure could be used as a worst case estimate.
You still need to handle the occasions where such a deadline is missed but you will be assured that the probability of missing the deadline is much reduced.

7 Conclusion

A simple system has been developed and presented that allows a measurement of real time performance and capability.
The results clearly show that , in the presence of system activity , the Real Time extensions to Linux do make a considerable difference. There is also a marked difference between the capability of Soft Real Time and Hard Real Time options currently available.

References

[1] Sources for System Benchmarks, http://www.anime.net/
goemon/benchmarks.html
http://www.byte.com/bmark/bmark.htm
http://www.math.vanderbilt.edu/
mayer/linux/results.html
http://www.kernelbench.org/
http://www.zentropix.com/
products/support/testdata.html

[2] Real Time Workshop 1999 , http://www.thinkingnerds.com/
projects/rtl-ws/presentations.html