Systemtap/Kprobes performance impact on N800

by Allan Bezerra, Bruna Moreira, Bruno Abinader and Roberto Santos
{allan.bezerra, bruna.moreira, bruno.abinader, roberto.santos}@openbossa.org
Instituto Nokia de Tecnologia [INdT] – Manaus – Brazil
Nokia Technology Institute – Manaus – Brazil

Change History

Issue	Date	Handled by	Comments
0.1	04/28/2008	Roberto Santos, Bruno Abinader	Initial version.
0.2	05/05/2008	Roberto Santos, Bruno Abinader	Remaing fixes to publish this report to the community.
0.3	13/05/2008	Bruno Abinader	Fixed kprobes source code on Micro Benchmark and wrong info about System Benchmark.

Contents

1 Introduction

Systemtap [1] provides a simple command line interface and scripting language for writing instrumentation for a live running kernel. It uses Kprobes for dynamic kernel instrumentation.

The objective of this report is to present the results over the elapsed time and the comparison of performance with Kprobes [3], Systemtap and Oprofile [5, 7].

This report is organized as follows: Section 2 shows Micro Benchmark and the setup environment for testing. System benchmark description and the setup environment for testing is in Section 3. Section 4 presents the measurement results, and at the end our conclusions.

2 Micro Benchmark

Based on Masami Hiramatsu's measurements [5], we have done the same experiment on N800. We attached a micro benchmark (`u-009` test from Sony test case [3]) to measure the processing time of multiple calls to `gettimeofday()`. This micro benchmark repeats `gettimeofday` system call for 10 (or another user-specified period) seconds and counts the number of calls. After that, it calculates average processing time per call.

2.1 Testing environment

The experiments were performed on N800 board, ARM architecture, loaded with kernel from ITOS2008 (Chinook 4.0.1) [8] release, kernel package kernel-source-rx-34_2.6.21, Systemtap (snapshot 20080405, elfutils-0.131). This kernel source was modified to attempt four configurations described bellow.

The following configured kernels were used:

- ITOS release with nokia_2420_defconfig, adopted as **DEFAULT**;
- Kprobes enabled not used;
- Kprobes enabled, with a probe at the top of do_gettimeofday();
- Systemtap: Kprobes environment plus systemtap package;
- Oprofile enabled, with and oprofile-common_0.9.3.1-osso1 package installed on target and oprofiled running on background [12].

Target for ARM, kernel image, modules and user space programs were compiled on Scratchbox 1.0 ARMEL target, cs2005q3.2-glibc2.5-arm compiler, loaded with a recent ITOS/maemo rootstrap filesystem (ARM rootstrap). We measured the overhead on linux-2.6.21 with five kernel configurations as described above.

2.2 Measurement Results

We used the *u-009* C program and *test.sh* script to collect the data in all environments. They are shown below:

```
INdT-N800:~/tests-stap# cat u-009.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/resource.h>
int main(int argc, char *argv[])
{
        struct timeval tv, otv;
        int sec = 10;
        unsigned long count;
        struct rusage usage;
        if (argc == 2)
                sec = atoi(argv[1]);
        gettimeofday(&otv, NULL);
        count = 0;
        do {
                gettimeofday(&tv, NULL);
                count++;
        } while ((sec - (tv.tv_sec - otv.tv_sec)) * 1000 >
                  (tv.tv_usec - otv.tv_usec) / 1000);
        getrusage(RUSAGE_SELF, &usage);
        tv = usage.ru_stime;
        tv.tv_sec += usage.ru_utime.tv_sec;
        tv.tv_usec += usage.ru_utime.tv_usec;
        fprintf(stderr,
                "gettimeofday was called %u times per %d sec: %f nsec per call"
                "\n", count, sec,
                      (float)(tv.tv_sec * 1000 * 1000 + tv.tv_usec) / (count /
1000));
        fprintf(stderr,
                "gettimeofday was called %u times per %d sec: %f usec per call"
                "\n", count, sec,
                (float)(tv.tv_sec * 1000 * 1000 + tv.tv_usec) / (count));
        return 0;
}
```

```
INdT-N800:~/tests-stap# cat test.sh
#!/bin/sh
for i in `seq 1 10`;
do
     /root/u-009
   sleep $1
done
```

We ran *test.sh* twice on each environment, collecting 20 samples in order to calculate average time on each of them.

2.2.1 Measurements on ITOS Release Kernel (not probed)

The sample output for a kernel not probed:

```
INdT-N800:~/tests-stap# ./test.sh
gettimeofday was called 5108818 times per 10 sec: 1959.242798 nsec per call
gettimeofday was called 5108818 times per 10 sec: 1.958929 usec per call
gettimeofday was called 5114240 times per 10 sec: 1956.944092 nsec per call
gettimeofday was called 5114240 times per 10 sec: 1.956852 usec per call
gettimeofday was called 5118601 times per 10 sec: 1955.414673 nsec per call
gettimeofday was called 5118601 times per 10 sec: 1.955185 usec per call
gettimeofday was called 5125648 times per 10 sec: 1952.743774 nsec per call
gettimeofday was called 5125648 times per 10 sec: 1.952497 usec per call
gettimeofday was called 5128365 times per 10 sec: 1951.601440 nsec per call
gettimeofday was called 5128365 times per 10 sec: 1.951463 usec per call
gettimeofday was called 5132964 times per 10 sec: 1950.080322 nsec per call
gettimeofday was called 5132964 times per 10 sec: 1.949714 usec per call
gettimeofday was called 5137888 times per 10 sec: 1948.182251 nsec per call
gettimeofday was called 5137888 times per 10 sec: 1.947845 usec per call
gettimeofday was called 5103663 times per 10 sec: 1961.162476 nsec per call
gettimeofday was called 5103663 times per 10 sec: 1.960908 usec per call
gettimeofday was called 5108263 times per 10 sec: 1959.242798 nsec per call
gettimeofday was called 5108263 times per 10 sec: 1.959142 usec per call
gettimeofday was called 5113376 times per 10 sec: 1957.326782 nsec per call
gettimeofday was called 5113376 times per 10 sec: 1.957183 usec per call
```

2.2.2 Measurements on ITOS Release Kernel (Kprobes)

In order to use Kprobes, it was necessary to fix some bugs that are explained on [2]. We have used a modified version of the *k-009* kernel module that uses both kprobe and kretprobe in order to insert a probe at the beginning and end of *do*_gettimeofday function:

Here is the code used for k-009.c:

```
/*
 *  k-009.c - A Kprobe with a probe point to the kernel function
 *  do_gettimeofday, with the empty pre-handler and post-handler
 *  which can be used to measure the Kprobes overhead.
 *
 *  Copyright 2007 Sony Corp.
 *
 *  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; version 2 of the License.
 *
 *  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
 */

#include <linux/module.h>
```

```
#include <linux/init.h>
#include <linux/kprobes.h>
#include <linux/kallsyms.h>

static struct kprobe k_009_kp1;

static int ret_handler(struct kretprobe_instance *ri, struct pt_regs *p)
{
      return 0;
}

static struct kretprobe k_009_kp1_return = {
      .handler = ret_handler,
      .maxactive = 100
};

static void __exit k_009_exit_probe(void)
{
      printk("\nModule exiting ");
      printk("from gettimeofday\n");
      unregister_kprobe(&k_009_kp1);
      unregister_kretprobe(&k_009_kp1_return);
}

static int k_009_pre_handler(struct kprobe *k_009_kp1, struct pt_regs *p)
{
      return 0;
}

static int k_009_post_handler(struct kprobe *k_009_kp1,
                        struct pt_regs *p, unsigned long flags)
{
      return 0;
}

static int __init k_009_init_probe(void)
{
      printk("\nInserting the kprobe at do_gettimeofday\n");

      /* Registering a kprobe */
      k_009_kp1.pre_handler = (kprobe_pre_handler_t) k_009_pre_handler;
      k_009_kp1.post_handler = (kprobe_post_handler_t) k_009_post_handler;
      k_009_kp1.addr = k_009_kp1_return.kp.addr =
          (kprobe_opcode_t *) kallsyms_lookup_name("do_gettimeofday");

      if ((k_009_kp1.addr == NULL) || (k_009_kp1_return.kp.addr == NULL)) {
            printk("kallsyms_lookup_name could not find address "
                   "for the specified symbol name\n");
            return 1;
      }

      printk("\nAddress where the kprobes are \n"
             "going to be inserted - %p \n", k_009_kp1.addr);
      register_kprobe(&k_009_kp1);
      register_kretprobe(&k_009_kp1_return);

      return 0;
}

module_init(k_009_init_probe);
```

```
module_exit(k_009_exit_probe);

MODULE_DESCRIPTION("Kprobes test module");
MODULE_LICENSE("GPL");
```

```
INdT-N800-:~/tests-stap# insmod k-009.ko
[ 230.562500]
[ 230.562500] Inserting the kprobe at do_gettimeofday
[ 230.578125]
[ 230.578125] Address where the kprobes are
[ 230.578125] going to be inserted - c0062138

INdT-N800-:~/tests-stap# ./test.sh 2
gettimeofday was called 1958188 times per 10 sec: 5047.401367 nsec per call
gettimeofday was called 1958188 times per 10 sec: 5.046917 usec per call
gettimeofday was called 2000757 times per 10 sec: 4996.093262 nsec per call
gettimeofday was called 2000757 times per 10 sec: 4.994203 usec per call
gettimeofday was called 1986078 times per 10 sec: 5035.246094 nsec per call
gettimeofday was called 1986078 times per 10 sec: 5.035048 usec per call
gettimeofday was called 1986170 times per 10 sec: 5039.180176 nsec per call
gettimeofday was called 1986170 times per 10 sec: 5.038749 usec per call
gettimeofday was called 1985961 times per 10 sec: 5041.718750 nsec per call
gettimeofday was called 1985961 times per 10 sec: 5.039279 usec per call
gettimeofday was called 1998086 times per 10 sec: 5008.915039 nsec per call
gettimeofday was called 1998086 times per 10 sec: 5.008699 usec per call
gettimeofday was called 1919487 times per 10 sec: 5215.118164 nsec per call
gettimeofday was called 1919487 times per 10 sec: 5.213795 usec per call
gettimeofday was called 1989930 times per 10 sec: 5031.579590 nsec per call
gettimeofday was called 1989930 times per 10 sec: 5.029228 usec per call
gettimeofday was called 1982710 times per 10 sec: 5045.408691 nsec per call
gettimeofday was called 1982710 times per 10 sec: 5.043602 usec per call
gettimeofday was called 1983284 times per 10 sec: 5042.863770 nsec per call
gettimeofday was called 1983284 times per 10 sec: 5.042142 usec per call

INdT-N800-:~/tests-stap# rmmod k-009.ko
[ 238.070312]
[ 238.070312] Module exiting from gettimeofday
```

2.2.3 Measurements with ITOS Release Kernel with Kprobes and Systemtap

On Scratchbox, we must have systemtap package installed, kernel source and symbolic links as mentioned in Systemtap documentation (<systemtap-source>/README).

```
[sbox-armel]:~# cat stap_gettimeofday_systemtap.stp
probe kernel.function("do_gettimeofday") { }
probe kernel.function("do_gettimeofday").return { }
[sbox-armel]:~# stap -v -p 4 -k -m stap_gettimeofday stap_gettimeofday.stp -r
2.6.21-omap1
Warning: using '-m' disables cache support.
Pass 1: parsed user script and 53 library script(s) in 2510usr/10sys/2577real
ms.
Pass 2: analyzed script: 1 probe(s), 1 function(s), 0 embed(s), 0 global(s) in
```

```
940usr/20sys/984real ms.
Pass 3: translated to C into "/var/tmp/[temp-dir]/stap_gettimeofday.c" in
10usr/0sys/7real ms.
Pass 4: compiled C into "stap_gettimeofday.ko" in 2600usr/610sys/3382real ms.
Keeping temporary directory "/var/tmp/[temp-dir]"
```

Now, just copy the /var/tmp/[temp-dir]/stap_gettimeofday.ko to the target.

Note: the ***-m*** argument renames the kernel module produced by stap command from a default string stap-<random_number>.ko to, for example, a more specific one kprobes-test.ko. Cache support for this operation is related to the results systemtap internal caching, thus not related to kernel cache operations and others. More information about this can be found on[9]. A good explanation about systemtap internal cache utilization can be found on [10].

On the target, we need to install just the systemtap package (binaries) in order to run the created modules.

```
INdT-N800-:~/tests-stap# staprun ./stap_gettimeofday_systemtap.ko > /dev/null &

INdT-N800-:~/tests-stap# ./test.sh 2
gettimeofday was called 1621074 times per 10 sec: 6173.850586 nsec per call
gettimeofday was called 1621074 times per 10 sec: 6.173569 usec per call
gettimeofday was called 1622532 times per 10 sec: 6170.044434 nsec per call
gettimeofday was called 1622532 times per 10 sec: 6.168021 usec per call
gettimeofday was called 1604896 times per 10 sec: 6239.284180 nsec per call
gettimeofday was called 1604896 times per 10 sec: 6.235801 usec per call
gettimeofday was called 1611172 times per 10 sec: 6212.173828 nsec per call
gettimeofday was called 1611172 times per 10 sec: 6.211511 usec per call
gettimeofday was called 1612616 times per 10 sec: 6208.320312 nsec per call
gettimeofday was called 1612616 times per 10 sec: 6.205949 usec per call
gettimeofday was called 1610277 times per 10 sec: 6216.032227 nsec per call
gettimeofday was called 1610277 times per 10 sec: 6.214963 usec per call
gettimeofday was called 1615800 times per 10 sec: 6196.787598 nsec per call
gettimeofday was called 1615800 times per 10 sec: 6.193719 usec per call
gettimeofday was called 1617274 times per 10 sec: 6189.123047 nsec per call
gettimeofday was called 1617274 times per 10 sec: 6.188075 usec per call
gettimeofday was called 1614875 times per 10 sec: 6200.626953 nsec per call
gettimeofday was called 1614875 times per 10 sec: 6.197267 usec per call
gettimeofday was called 1620446 times per 10 sec: 6177.661621 nsec per call
gettimeofday was called 1620446 times per 10 sec: 6.175961 usec per call
```

2.2.4 Measurements with Oprofile

For this scenario, it was used a specific kernel image was used to run Oprofile [12]. We created the *oprofile-gettime.sh* script to start tracing and call the u-009 program:

```
INdT-N800-:~/tests-stap# cat oprofile-gettime.sh
#!/bin/sh
opcontrol --init; opcontrol --status
opcontrol --reset
opcontrol --start --separate=lib --no-vmlinux
```

```
./test.sh 2
opcontrol -h
```

```
[output for the command "test.sh 2" inside oprofile-gettime.sh]
Daemon not running
Separate options: library
vmlinux file: none
Image filter: none
Call-graph depth: 0
Using default event: CPU_CYCLES:100000:0:1:1
Using 2.6+ OProfile kernel interface.
Using log file /var/lib/oprofile/samples/oprofiled.log
Daemon started.
Profiler running.
gettimeofday was called 4975966 times per 10 sec: 2011.620483 nsec per call
gettimeofday was called 4975966 times per 10 sec: 2.011230 usec per call
gettimeofday was called 4798866 times per 10 sec: 2036.981445 nsec per call
gettimeofday was called 4798866 times per 10 sec: 2.036614 usec per call
gettimeofday was called 4800458 times per 10 sec: 2036.132690 nsec per call
gettimeofday was called 4800458 times per 10 sec: 2.035939 usec per call
gettimeofday was called 4801833 times per 10 sec: 2035.708618 nsec per call
gettimeofday was called 4801833 times per 10 sec: 2.035356 usec per call
gettimeofday was called 4803839 times per 10 sec: 2036.487427 nsec per call
gettimeofday was called 4803839 times per 10 sec: 2.036132 usec per call
gettimeofday was called 4802238 times per 10 sec: 2038.538574 nsec per call
gettimeofday was called 4802238 times per 10 sec: 2.038438 usec per call
gettimeofday was called 4916076 times per 10 sec: 2035.763184 nsec per call
gettimeofday was called 4916076 times per 10 sec: 2.035732 usec per call
gettimeofday was called 4804480 times per 10 sec: 2037.689819 nsec per call
gettimeofday was called 4804480 times per 10 sec: 2.037486 usec per call
gettimeofday was called 3547717 times per 10 sec: 2713.560547 nsec per call
gettimeofday was called 3547717 times per 10 sec: 2.713012 usec per call
gettimeofday was called 4915630 times per 10 sec: 2036.177368 nsec per call
gettimeofday was called 4915630 times per 10 sec: 2.035917 usec per call
Stopping profiling.
Killing daemon.
```

3 System Benchmark

We used GtkPerf tool [7], version 0.40, for system benchmark measurements. GtkPerf is an application designed to test GTK+ performance. The point is to create a common testing platform to run predefined GTK+ widgets (opening comboboxes, toggling buttons, scrolling text etc.), thus defining the speed of device/platform.

More detailed information about each test can be found inside the GtkPerf documentation (gtkperf-0.x/README). Note that some of the tests have random behaviors. These are the tests:

- GtkDrawingArea - Lines
- GtkDrawingArea - Circles
- GtkDrawingArea - Text
- GtkDrawingArea – Pixbuf

3.1 Testing Environment

The experiments were performed on N800 board, ARM architecture, ITOS2008 (Chinook 4.0.1) release. The following environments were setup on top of a kernel 2.6.21 release source tree:

- ITOS kernel release with nokia_2420_defconfig (NOT PROBED);
- Kprobes enabled and not used;
- Systemtap: Kprobes environment plus systemtap module;
- Oprofile (kernel image: zImage-oprofile-rx-34-200749) [12];

The Kprobes ARM implementation used was kprobes-arm-motorola-2.6.16.patch [3]. Kernel image, modules and user space programs were compiled on Scratchbox 1.0 ARMEL target, cs2005q3.2-glibc2.5-arm compiler, loaded with a recent ITOS rootstrap filesystem (ARM rootstrap, week 33).

3.2 Measurements results

Gtkperf was ran 5 times on each environment (Section 3.1), collecting 500 samples for each time to calculate average time on each of them. The command line used is shown below:

```
INdT-N800-:~/tests-stap# gtkperf -c 500 -a
GtkPerf 0.40 - Starting testing: Wed Feb  1 14:37:20 2008

GtkEntry - time:  4.17
GtkComboBox - time: 73.71
GtkComboBoxEntry - time: 60.85
GtkSpinButton - time: 11.43
GtkProgressBar - time:  4.89
GtkToggleButton - time: 34.73
GtkCheckButton - time: 34.57
GtkRadioButton - time: 38.03
GtkTextView - Add text - time: 209.94
GtkTextView - Scroll - time: 101.84
GtkDrawingArea - Lines - time: 52.55
GtkDrawingArea - Circles - time: 84.15
GtkDrawingArea - Text - time: 100.31
GtkDrawingArea - Pixbufs - time: 14.53
 ---
Total time: 825.78

Quitting..
```

The lines with GtkDrawingArea were left out this report because they use random() values and this makes the results unreliable.

The setup for the SystemTap-Kprobes environment the following script used was:

```
INdT-N800-:~/tests-stap# cat stap_tapsets.stp
probe vm.* { log ("vm") }
probe process.* { log ("process") }
probe signal.* { log ("signal") }
probe syscall.* { log ("syscall") }

INdT-N800-:~/tests-stap# stap stap_tapsets.stp &
```

4 Overall results

4.1 Results for Micro Benchmark

A simple average from processing time is used in Table 4.1.1 and Chart 4.1.1:

Environment setup	DEFAULT	KPROBES SET NOT USED	KPROBES SET AND USED	SYSTEMTAP PLUS KPROBES	oprofile
current	1.942035	1.968863	5.0491662	5.973947	2.052599

Table 4.1.1: results are shown in microseconds.

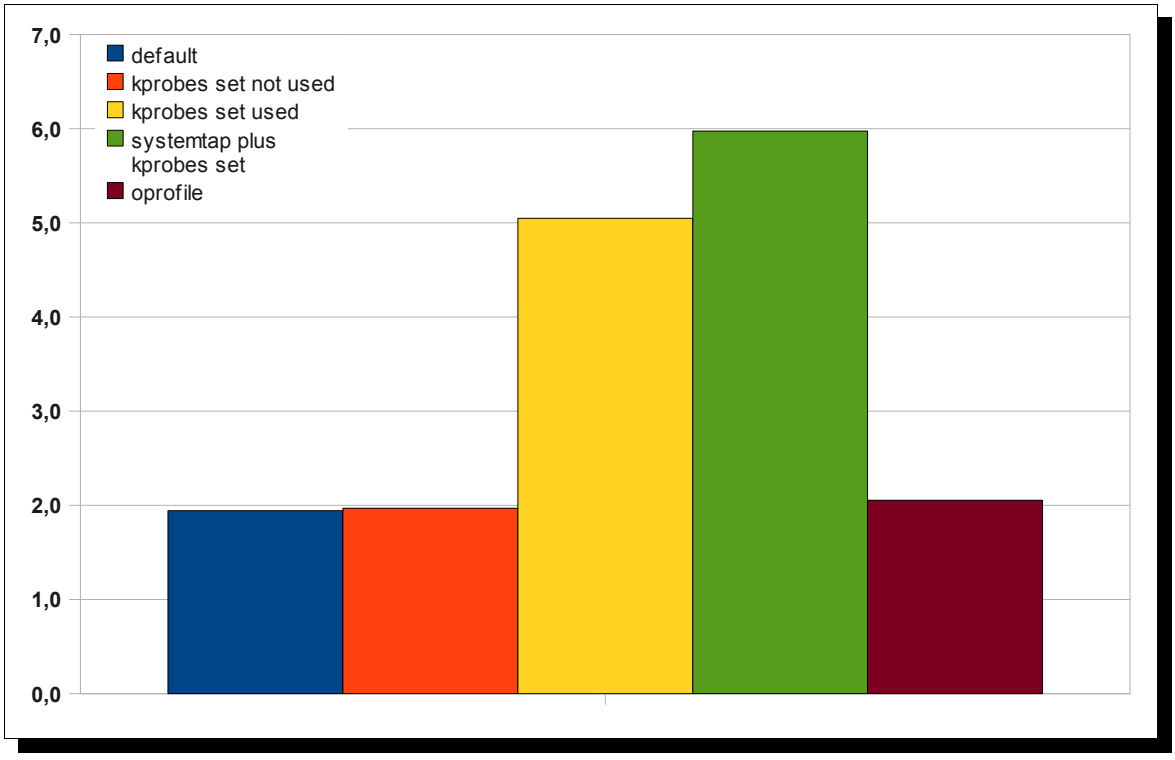

Chart 4.1.1.

The Chart 4.1.1 refers to the line **microseconds** from Table 4.1.1. In the Table 4.1.2 shows the overhead in percentage (minor is better) compared to default kernel configuration:

Environment setup	KPROBES SET NOT USED	KPROBES SET AND USED	SYSTEMTAP PLUS KPROBES	oprofile
overhead (%)	+ 1.38	+ 159.99	+207.61	+ 5.56

Table 4.1.2.

4.2 Results for System Benchmark

For this benchmark we ran twice for each kernel scenario (section 3.1) and calculated an average from elapsed time. The results are presented in next tables and charts.

TEST	DEFAULT	KPROBES SET NOT USED	SYSTEMTAP PLUS KPROBES	OPROFILE
GtkEntry	3,69	3,73	6,71	3,63
GtkSpinButton	10,46	10,33	17,77	10,38
GtkProgressBar	4,59	4,6	7,12	4,6
GtkComboBox	74,4	72,5	118,38	67,58
GtkComboBoxEntry	59,32	59,53	84,5	56,12
GtkToggleButton	32,37	32,26	43,88	32,32
GtkCheckButton	32,46	31,94	43,59	32,21
GtkRadioButton	35,31	35,08	48,52	35,41
GtkTextView - Add text	207,1	193,25	355,67	188,33
GtkTextView - Scroll	93,93	91,65	134,75	91,54

Table 4.2.1.

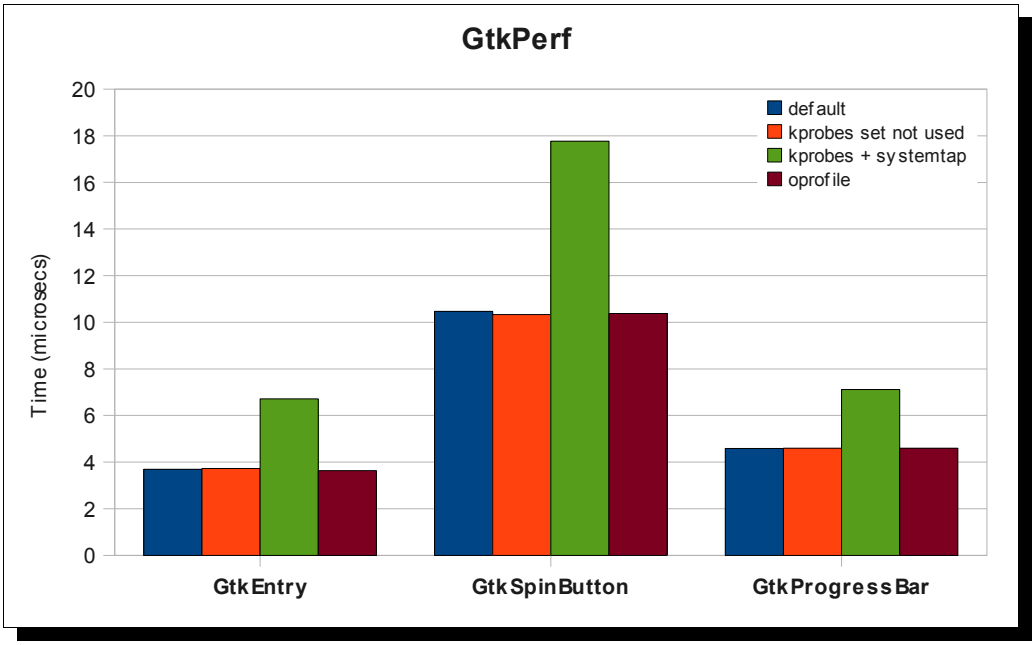

Chart 4.2.1.

These charts show the values represented graphically for better visualization of the elapsed time. The output values were separated in three charts and they refer to the Table 4.2.1.

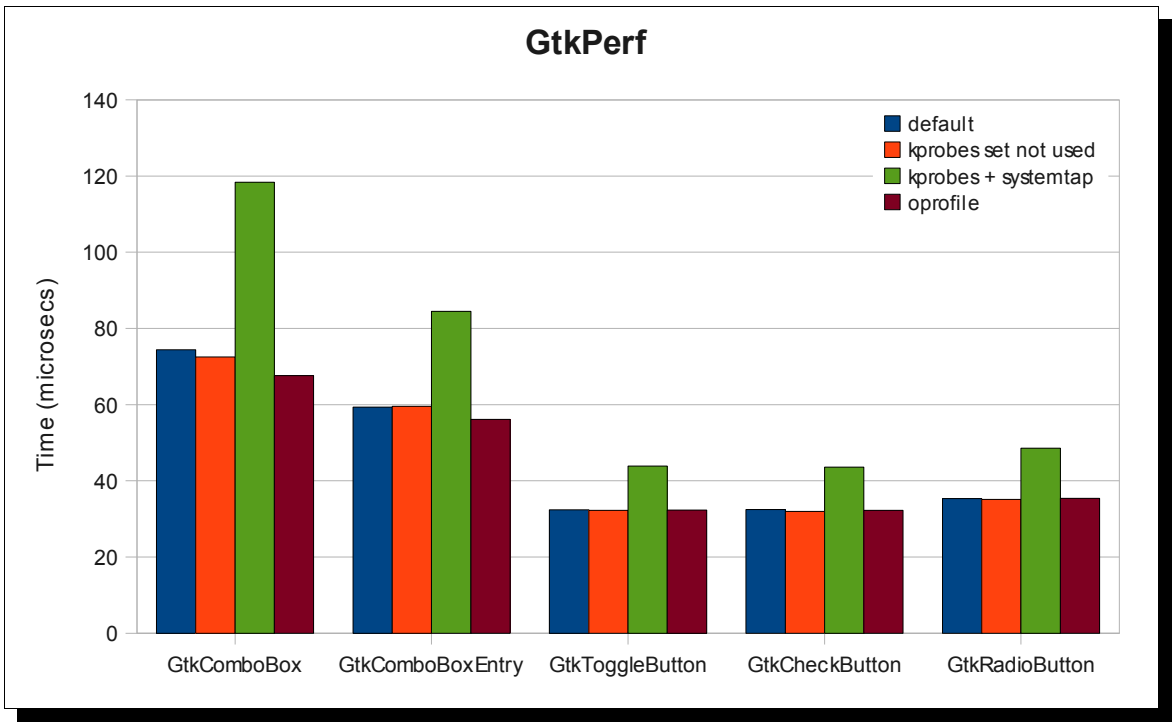

Chart 4.2.2.

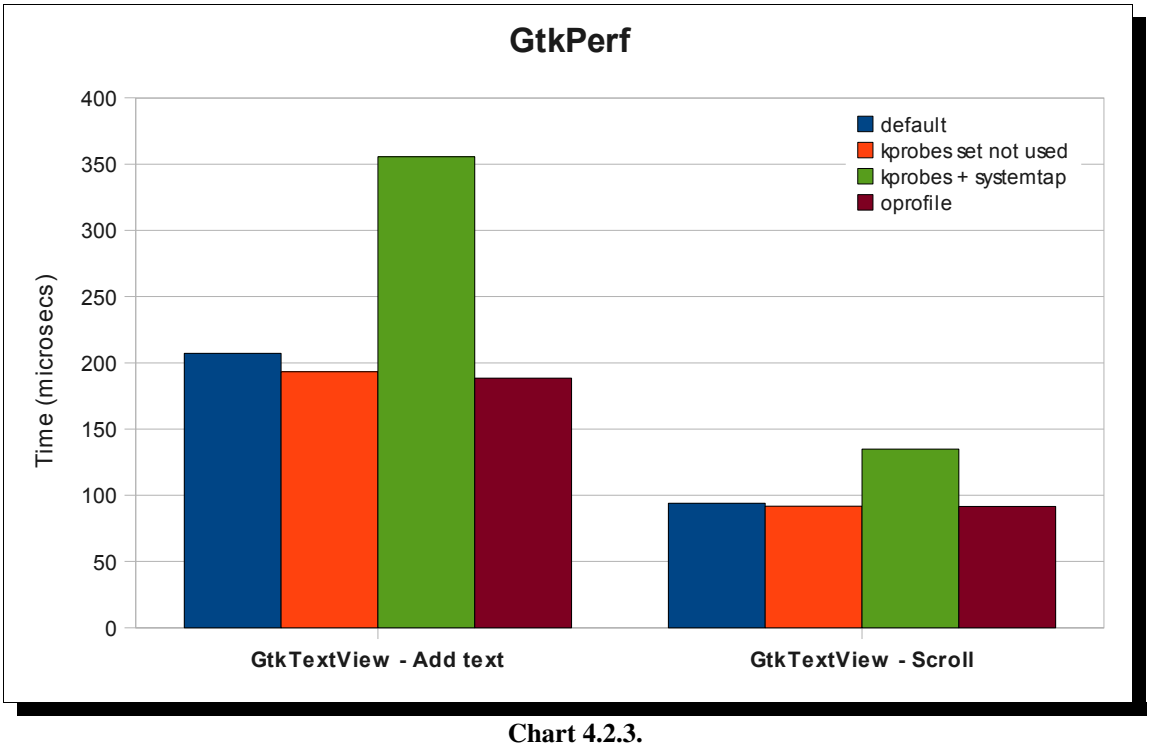

Chart 4.2.3.

Conclusions

In this report, we presented a performance evaluation impact of Systemtap and Kprobes for N800/N810 on ARM architecture, using two kinds of benchmarks processes: micro and system Benchmark. According to the micro benchmark result, it shows that Systemtap with Kprobes imposes a significant processing overhead when compared to ITOS2008 (Chinook 4.0.1) original release, Kprobes-only and Oprofile.

It seems that the additional “house-keeping” C code added in translated Systemtap scripts (see the `.c` output from systemtap) is responsible for such overhead. Additionally, when Kprobes is enabled in kernel compilation and the probes are not inserted at run-time, the processing time overhead is minimal. Oprofile has a similar behavior, when it is not enabled the overhead can be discharged.

About the System benchmark results, the kernel with Kprobes support enabled but not used has near zero overhead over an ITOS kernel. At last, Systemtap has once again shown the highest overhead on system benchmark. The relative difference is lower however, as it does not target a specific event as in the micro benchmark.

References

[1] Systemtap. URL: http://sourceware.org/systemtap/
[2] Howto Systemtap on Maemo, URL: http://sourceware.org/systemtap/wiki/SystemtapMaemoBenchmark
[3] Sony Kprobes ARM patches. URL: http://tree.celinuxforum.org/CelfPubWiki/PatchArchive?action=AttachFile&do=get&target=kprobes-arm-patches-2.6.16.24.tgz
[4] Oprofile, URL: http://maemo.org/development/tools/doc/oprofile
[5] Masami Hiramatsu's measurements. URL: http://sourceware.org/ml/systemtap/2006-q1/msg00318.html
[6] http://oprofile.sourceforge.net/news/
[7] GtkPerf website. URL: http://gtkperf.sourceforge.net/
[8] Kernel source: URL: http://repository.maemo.org/pool/os2008/free/source/k/kernel-source-rx-34/
[9] http://sourceware.org/bugzilla/show_bug.cgi?id=4281
[10] http://sourceware.org/bugzilla/show_bug.cgi?id=4281#c5
[11] Howto Systemtap on Maemo, URL: http://sourceware.org/systemtap/wiki/SystemtapMaemo
[12] Oprofile kernel. URL: http://repository.maemo.org/pool/chinook/free/r/rx-34-kernel/rx-34-kernel-oprofile_2.6.21.0-200749osso2_all.deb