Linux Kernel Hack Challenge Specification

Version 1.0.0

Linköping 2014-10-27

Table of contents

1 SCOPE

1.1 Identification

This is a specification for an execution time profiling of applications running under Linux.

2 SPECIFICATION

2.1 Background

The profiling aims to evaluate execution time for an application that in its target environment is subject to context switches by a Real-Time Operating System (RTOS). Such context switches implies the flushing and invalidation of L1 caches (no other caches are used), which results in slowed down execution - even if the time for the context switch itself is disregarded.

The profiling is intended to measure execution time in a Linux test environment which is a much cheaper and more accessible environment. The test environment consists of systems based on the same processor, and configured with the same clock frequencies, as the target. They are not running the RTOS, but a version of Linux OS, provided by the processor/test system manufacturer (Freescale).

There has been a failed attempt to perform the profiling described herein, which results of patches to the Linux kernel an supporting kernel modules can be found as reference for this contest. The patched kernel destroyed some of the characteristics for scheduling Linux, and the desired behavior in Acceptance test 6 in chapter 3.2.1 was destroyed by:

a) Intermittent spikes of time measurement
b) The first 50-80 us of profiling always executed faster than if not being interrupted.
c) The execution time for interrupts occurring after execution for the profiled application as reported as faster than the non-interrupted execution (which it principally should have been).
d) The profiling library often gave an error code that the result could not be trusted.

2.2 How do I win the contest?

The goal of the contest is to provide a solution that satisfies the Acceptance tests in chapter 3 by:

a) Finding another solution to the problem that was investigated in the first attempt
b) Correcting the patch of the scheduling in the kernel to provide real-time characteristics in a small time slot.

2.3 General Idea

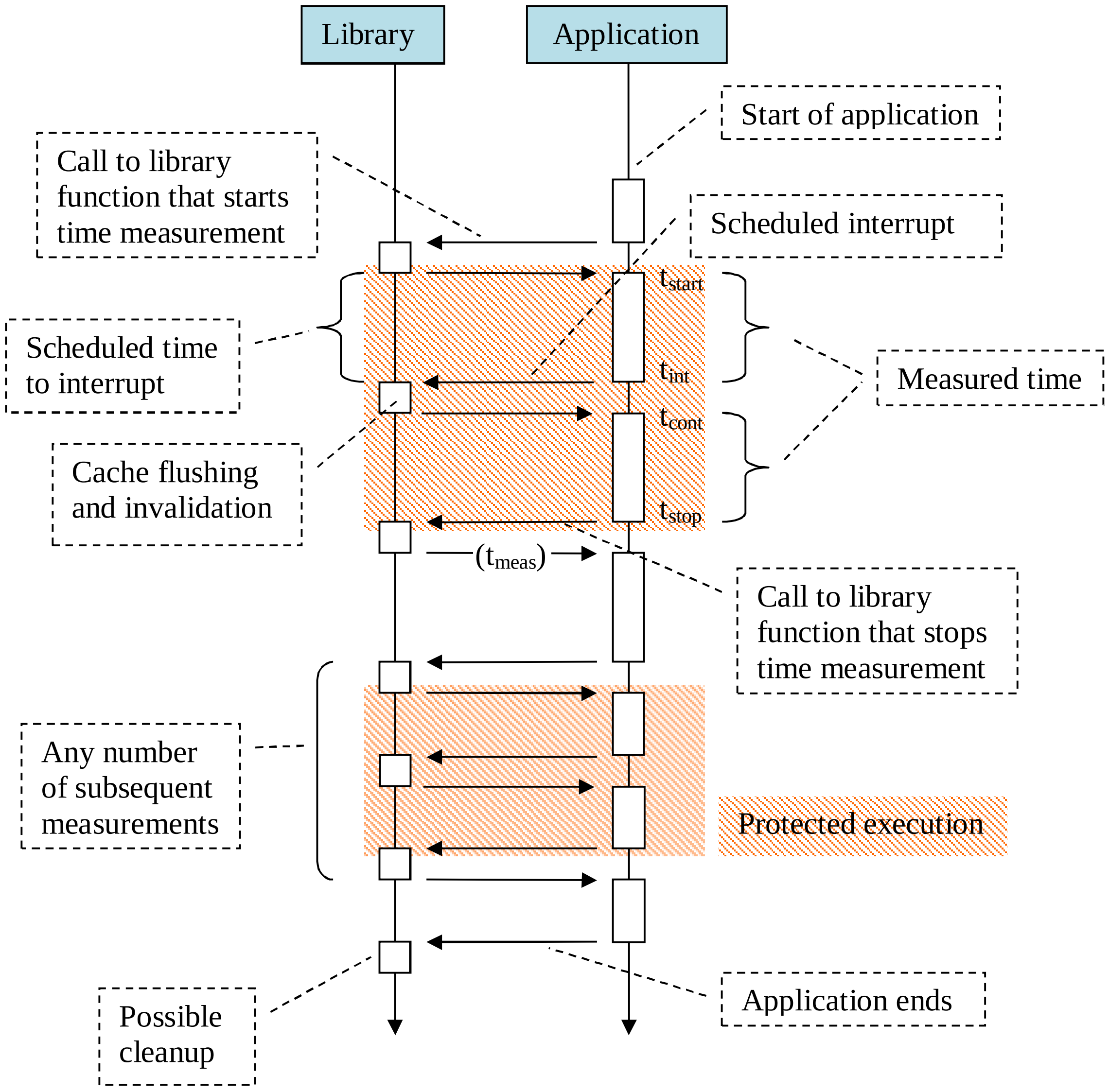
The profiling shall be used to measure the execution time between two points within an application, defined by calls to library functions supplied by a Profiling library. During the execution, the profiling shall simulate a context switch at a scheduled point in time, by interrupting the application and flushing/invalidating the L1 caches.

To accomplish accurate measurements, i.e. measurements not affected by the Linux OS doing other things in parallel, the profiling library (probably) needs to disable essential parts of Linux OS, such as interrupts, and/or use increased priority (or kernel mode) during the measurement. This gives “real-time” characteristics of the Linux kernel for this small time interval, and also means that a general timeout that aborts a hanging application under test may also be required, to ensure the possibility to regain remote control of the system without physical reboot of the system.

2.4 Principle

The application subject to the measurements consists of two main parts linked together; the actual application, which is available only as object code, and a test harness, which is used to set up the application part in the desired state before it triggers the measurement. When built, we assume the application to be linked to a profiling library containing the “start time measurement” and “stop time measurement” functions.

When the application is started, execution commences according to this picture. In the picture the two parts together are called “application” as it is not of interest from the profiling library how it is constituted.


Library
Application
Start of application
Call to library function that starts time measurement
Scheduled interrupt
t_{start}
Scheduled time to interrupt
t_{int}
Measured time
t_{cont}
Cache flushing and invalidation
t_{stop}
(t_{meas})
Call to library function that stops time measurement
Any number of subsequent measurements
Protected execution
Possible cleanup
Application ends

The call to "start time measurement" states the time to wait before *interrupting* the execution for a simulated context switch, where L1 cache flush invalidate is performed. It shall also be possible to have no interrupt scheduled. This call is also the signal to enter the protected execution mode, where the application is protected from other interrupts. Nothing else shall be allowed to execute during this time (Except "Cache flushing and invalidation" in picture), since that would affect caches, pipelines and so on, thus indirectly affecting execution speed.

The response to the "stop time measurement" call shall contain the result of the measurement. This call is also the signal to leave the protected execution mode with "real-time characteristics". It shall also be indicated by the response if the measurement is stopped before the scheduled cache flush and invalidation took place.

The measured time shall exclude the simulated context switch as such, so

$t_{meas} = (t_{stop} - t_{start}) - (t_{cont} - t_{int})$

The application may then start a new measurement, but there will only be one active at a time. The application is always *single-threaded*.

2.5 Details

Environment:

- Freescale processor with PowerPC e500v2 core
  Note: This is the platform where contest contributions will be evaluated. No such platform for contestants will be provided for the contest.
- Linux 2.6.32
- gcc compiler v 4.1.2

Interfacing:

- The application subject to the measurement is an executable for the test environment
- It is linked to a library that contains the start/stop calls
  - Start call takes the time to the scheduled interrupt, if any, as argument.
  - Stop call responds with the measured time. Depending on accuracy, it may also be interesting to get the actual time to the scheduled interrupt.

Suggested syntax:

```
void START_PROFILING_TIMER(long long int Interrupt_At_US, long long int General_Timeout_US)
int STOP_PROFILING_TIMER(long long int* Measured_Time_US)
```

Resolution/accuracy:

- In the range of microseconds, both for measurement and scheduling.

Note: The 2.6.32 kernel has been tested to provide us resolution of time.

Timing:

- Maximum length of one measurement is approx. 100 ms, typical length is considerably less (50 – 1000 us).

Delivery for contest:

- Source code in C
- Makefiles for building (and installing) the profiling library, if applicable

Acceptance:

- A less complex test application is provided for experimental purposes. It will also be used in acceptance test 6, see chapter 3.2.1.

3 ACCEPTANCE TEST PROCEDURE

The contributions to the *Linux Kernel Hack Challenge* contest will be subject to this acceptance test procedure.

The tests to evaluate the contributions will be performed on the above mentioned Freescale processor with PowerPC e500v2 core.

This section also describes characteristics that a *Kernel Hack Challenge* solution should comply to.

3.1 Installation

3.1.1 Acceptance test 1: Installation test

Preconditions: User manual and delivered source code and make files available.

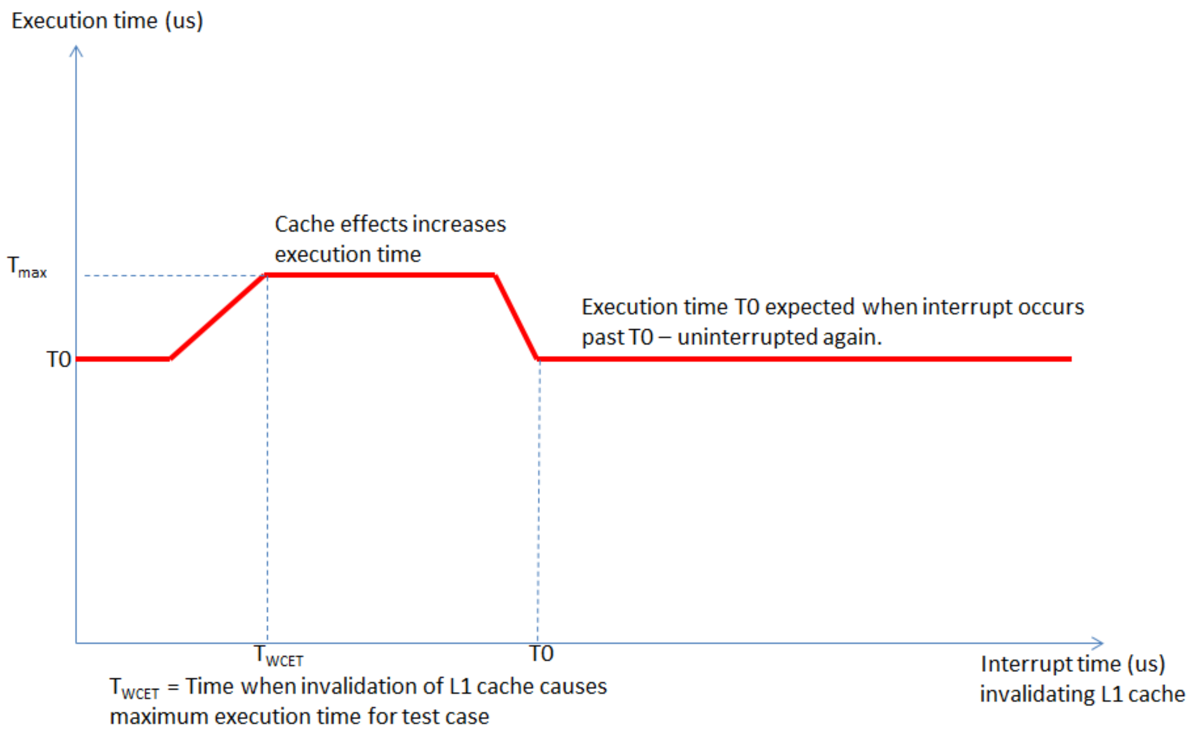
Acceptance test	**Result (Pass/Fail)**
Can the library be built using makefiles and source code (User manual)?	

3.2 System validation

3.2.1 Acceptance test 6: Profiling capability

Preconditions: Test application available; the application shall iterate over a 30 kB data array, first accessing an increasing amount of the data in each iteration, then a decreasing amount.

Acceptance test	**Result (Pass/Fail)**
1. Let the application flush and invalidate L1 cache. 2. Start Time measurement specifying interrupt to be performed after 1 us. 3. Call the iterating code sequence. 4. Stop Time measurement. 5. Report the measured time. 6. Repeat 2-5 above, each time increasing the specified time to interrupt by 1 us, until the interrupt time exceeds the execution time 7. Verify that the reported measured times form an adequate profile, see below.	



3.3 Time measurement

3.3.1 Acceptance test 2: No interrupt

Preconditions: A small application with known execution time exists.

Acceptance test	Result (Pass/Fail)
1. Let an application flush and invalidate L1 cache. 2. Start Time measurement, specifying no interrupt to be performed. 3. Call code sequence in small application with known execution time. 4. Stop Time measurement. 5. Repeat 1-4 above 10000 times. 6. Verify that: a. The time reported is consistent with the known reported time. b. All reported times are not fluctuating largely (stable on a +/- 2us basis)	

3.3.2 Acceptance test 3: Interrupt after execution end

This is a robustness test case verifying that the provided interrupt time for a time after execution end does not occur.

Preconditions: A small application with known execution time exists.

Acceptance test	Result (Pass/Fail)
1. Let the application flush and invalidate L1 cache. 2. Start Time measurement specifying interrupt to be performed after expected stop of time measurement. 3. Call code sequence in small application with known execution time. 4. Stop Time measurement. 5. Repeat 1-4 above 10000 times. 6. Verify that a. The time reported is consistent with the known reported time for uninterrupted application. b. All reported times are not fluctuating largely (stable on a +/- 2us basis)	

3.3.3 Acceptance test 4: Normal timed interrupt

Preconditions: A set of small applications with known execution time exists.

Criteria for evaluation: Measured times shall be unaffected by other processes execution, and within specified time intervals per test case below.

Acceptance test	Result (Pass/Fail)
1. Let the application flush and invalidate L1 cache. 2. Start Time measurement specifying interrupt to be performed after *INTERRUPT_TIME* us which is a time inside the expected execution time, in time interval $T_{start} – T_{stop}$. 3. Call code sequence in small application with known execution time. 4. Stop Time measurement. 5. Repeat 1-4 above 10000 times. 6. Verify that: a. The reported times are consistent (stable on a +/- 2us basis) b. The reported time is less than or equal to the sum of uninterrupted time measured and the interrupt time. $Tmeasured \leq Tuninterrupted + INTERRUPT\_TIME$ c. The reported time is larger than or equal to the uninterrupted time measured: $Tmeasured \geq Tuninterrupted$	

7. Repeat 1-4 above for scheduled interrupt time between 1 us and *Tuninterrupted* us in intervals in us range 8. Verify that: a. The profiling time measurement presents measurement data where reported times are less than or equal to the sum of uninterrupted time measured and the interrupt time: *Tmeasured < Tuniterrupted + INTERRUPT_TIME* b. The profiled execution times are consistent between different executions. Some time differences is expected but less than 2 us.	

3.3.4 Acceptance test 5: Application not returning from execution

This is a robustness test case verifying that the provided General Timeout for a non-returning application works as expected.

Preconditions: A small application with infinite loop.

Acceptance test	**Result (Pass/Fail)**
1. Let the application flush and invalidate L1 cache. 2. Start Time measurement specifying interrupt to be performed after 100 us and a specified *General Timeout*. 3. Call code sequence in small application with infinite loop. 4. Verify that the profiling library kills the application execution after approximately the *General Timeout* time. 5. Repeat 1-4 above 100 times. 6. Verify that: a. The profiling always kills the application execution.	