Unintrusively Measuring Linux Kernel Execution Times

Sven Schneider and Robert Baumgartl
Chemnitz University of Technology
09107 Chemnitz, Germany
{sven.schneider,robert.baumgartl}@informatik.tu-chemnitz.de

Abstract

We present a methodology to perform fine-grained cycle-accurate timing measurements crossing the user-kernel boundary. Special attention is payed not to deteriorate the results by the measurement process itself. Next, we apply our methodology to obtain execution timing for the system entry and exit paths on x86-based Linux systems. We compare and evaluate three different mechanisms, namely interrupts, call gates and the sysenter/sysexit facility. The measurements are performed on different processor platforms ranging from simple Intel Pentium to Pentium 4 with Hyper Threading and an Intel Itanium system. Our results indicate that timing for kernel entry and exit varies in a non-trivial way. Of the tested architectures, the AMD Athlon exhibits the most efficient behavior.

1 Introduction

The work described in this paper grew from a discussion on operating system efficiency. The question arose, how many time an operating system needs for the system entry and exit itself without paying attention to the actual system call functionality. Additionally, we wanted to find out, whether there exist timing differences between microprocessors of different architecture and age concerning the system entry and exit paths. Finally, we are also interested in a comparison between the different mechanisms available for system entry.

Of course, measuring execution times in user or kernel mode is as easy as reading the time stamp counter at appropriate positions. The transition between both modes is more difficult to analyze. Therefore, we present a methodology to obtain fine-grained performance numbers. For a number of reasons we solely concentrate on Linux.

2 Related Work

There are a variety of profiling tools and methodologies available, yet none of them suited our needs perfectly. For the profiling of the system entry and exit paths a very fine-grained and precise method is required. Hence, the measurement function should not use the system entry and exit paths itself (e. g. by using system calls) and the profiling data should be gathered in a way which influences the timing of the system entry and exit paths as little as possible.

The Linux Tracing Toolkit [2] is a universal profiling and event tracking tool. It provides timing measurements for both user mode (via a system call) and kernel mode (via a normal function call) functionality. To analyze the collected data, a set of powerful tools is available. Unfortunately, user mode measurement points can only be established using a system call, which violates our requirements.

Another approach could be the use of a kernel profiler kit. The Profiling in Linux HOWTO [1] describes the different general and special purpose profilers thoroughly. Unfortunately, every kernel profiler either (a) uses statistical profiling which is far too inaccurate for our project, or (b) lacks the opportunity to profile the interaction between user and kernel mode. Therefore, kernel profilers were not regarded further.

A technique called *Code Splicing* is described in [5]. It is based on dynamically overwriting single instructions with a jump to the instrumentation code, the overwritten instruction and a jump back. This approach is very challenging on variable-length instruction architectures as IA-32, because it is hard to predict in advance how much of an instruction the jump actually overwrites. As a consequence, it has been implemented for the UltraSPARC architecture only. Additionally, this technique is not useable when switching between user and kernel mode.

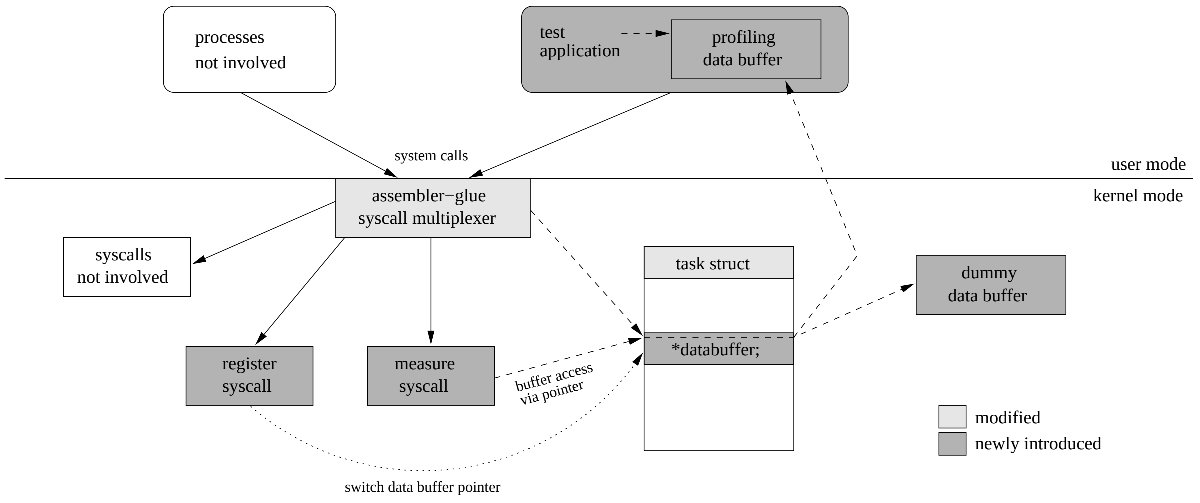

FIGURE 1: *Experimental Setup*

The DeBox project [3] provides a very simple interface for the application. The application simply registers a buffer for data collection. Every time it leaves kernel mode after that registration, the internally gathered profiling data is copied into the buffer. When no buffer had been registered no copying is performed. Data is passed in-band without relying on extra system calls. DeBox is available for FreeBSD only, therefore it is as such inapplicable too, but its concept is closest to our own ideas. The data copying can be eliminated by directly accessing the buffer from kernel mode (as our method does, cf. section 3.1).

3 Measurement Methodology

Our measurement methodology of the system entry and exit paths should provide a very fine-grained resolution. Furthermore, it is important that gathering the profiling data does influence execution timing as little as possible. To avoid disturbance of the branch prediction, no conditional jumps are allowed within the instrumentation code. To eliminate cache influences, measurements were repeated many times and the instrumentation code was reduced to a minimum size.

3.1 System Structure

Figure 1 shows the experimental setup. The test application registers a profiling data buffer to the kernel via a newly introduced system call. To that aim, the task struct has been extended by an additional pointer ("databuffer") which either points to the registered buffer or a dummy buffer. Unregistering is done by "registering" a null pointer. The register system call switches between a dummy data buffer and the buffer provided by the test application. The default buffer for a process is the dummy buffer. At selected measurement points the current time stamp is written at the appropriate position within the profiling data buffer. For the actual measurement another system call has been introduced which does not do anything besides writing the time stamps. The test application is responsible for storing the collected data in user space before issuing the next syscall after the measurement, because time stamps are collected for *every* syscall (otherwise, a conditional statement would be necessary on system entry which would influence the branch prediction unit). All other processes also generate time stamps on every syscall, which are written into the dummy data buffer. The dummy buffer is never read, therefore, no locking for consistency reasons is necessary.

The test application allocates a large buffer to store a set of measurements and the (small) profiling data buffer, which is registered to the kernel. Then, a single measurement consisting of up to eight individual time stamps is taken by issuing the *measure syscall*. On return the collected time stamps are copied into the large buffer to avoid losing them by the following system call. This is repeated until the large buffer is filled. The contents of the large buffer represents the timing data of the system entry and exit paths of a particular system configuration.

3.2 Measurement Points

Time stamps are taken at the following locations (as shown in figure 2):

MP1 "enter_pre_sci" Directly before system call invocation. Work which is done before the

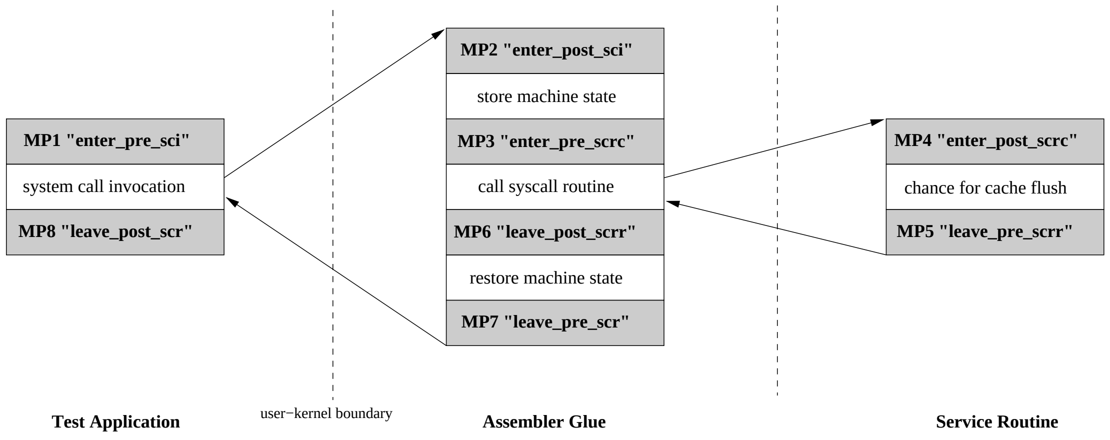

FIGURE 2: *Placement of Measurement Points*

privilege level changes is counted as additional time needed for the switch itself.

MP2 "enter_post_sci" As soon as possible after entering kernel mode. Some architectures need some post processing before the instrumentation can be executed. This post processing time is counted as additional time needed for the switch.

MP3 "enter_pre_scrc" Directly before calling the system call service routine.

MP4 "enter_post_scrc" The first . . .

MP5 "leave_pre_scrr" . . . and the last instruction in the service routine.

MP6 "leave_post_scrr" The first instruction at the location where the service routine returns to.

MP7 "leave_pre_scr" As late as possible before the privilege level changing instruction where the kernel is left.

MP8 "leave_post_scr" The first instruction in the test application after returning from the system call.

3.3 Implementation Issues

Measurement points 1 and 8 are user mode instrumentations (as inline assembler for a C function) and are located in the code of the test application. The points 2, 3, 6 and 7 are located in the assembler glue and therefore are real assembler macros. Finally, points 4 and 5 reside in the measurement system call. Hence, they are implemented as inline assembler within a C function (and actually use the same macro as points 1 and 8).

To ease porting the methodology to different CPU architectures, the instrumentation code is divided into an architecture-independent and an architecture-dependent part. The latter consists mainly of the measurement point implementation. As already mentioned, there are two different versions, depending on the position of the point. For example, the macro for points 2, 3, 6 and 7 for the IA32 architecture is implemented as follows:

```
#define PROC_LARGE_MP(x) \
    pushl %eax ;\
    pushl %ebx ;\
    pushl %ecx ;\
    pushl %edx ;\
    xorl %eax, %eax ;\
    cpuid ;\
    GET_CURRENT(%ebx) ;\
    movl SCMI_DATA_OFFSET(%ebx),%ebx ;\
    rdtsc ;\
    movl %edx,((x)+4)(%ebx) ;\
    movl %eax,(x)(%ebx) ;\
    xorl %eax,%eax ;\
    cpuid ;\
    popl %edx ;\
    popl %ecx ;\
    popl %ebx ;\
    popl %eax
```

Due to possible instruction reordering, reading the time stamp counter must be protected by serializing instructions, hence the surrounding `cpuid`. Register `ebx` holds the pointer with the profiling data buffer's address. The parameter `x` of the macro is the individual position of the time stamp within the profiling data buffer. The macro is inserted at appropriate locations in `arch/i386/entry.S` .

Besides IA-32, we also implemented our methodology for the Intel IA-64 architecture. More information concerning both implementations can be found in [4].

3.4 Kernel Entry/Exit Variants

The standard system call mechanism of Linux 2.4.x uses the interrupt *0x80* for privilege level change, then stores all registers on the stack, multiplexes the service routine, handles signals and rescheduling, restores the registers and finally returns to user mode. To get an idea how the accompanying functionality influences the time to switch to the service routine and back, different variants of the system entry and exit were implemented (cf. table 1).

	intr	call gate	sep
Standard Work	int80	cg1	sep1
skip *-handling code	int81	cg2	sep2
Service Routine only	int82	cg3	sep3

TABLE 1: *Overview of Kernel Entry/Exit Variants*

A first aspect was to reduce the work on system entry and exit. Therefore, besides the "full-featured" entry path (Standard Work), two variants with reduced functionality were implemented. The first one eliminates signal and scheduling handling code (skip *-handling Code) and the second one calls the service routine directly (Service Routine only). Even system call multiplexing has been removed from that latter variant.

The Intel IA-32 architecture provides three different mechanisms for changing the privilege level, namely by interrupt (intr), by call gates (call gate) and by sysenter/sysexit mechanism (sep). The three mechanisms were combined with the three implementation complexity variants resulting in nine different system configurations.

3.5 Experimental Platforms

A total of six different platforms were analyzed:

- IA-32

  P1 – Intel Pentium MMX, 250 MHz

  P2 – Intel Pentium II, 400 MHz

  P3 – Intel Pentium III, 1.1 GHz

  P4 – Intel Pentium 4 with HT, 2.8 GHz,

  At – AMD Athlon, 1.0 GHz

- IA-64

  It – Intel Itanium 2, 900 MHz

The measurements for the IA-32 were based on Linux 2.4.24 and 2.4.30 kernels.

The IA-64 version is based on Linux 2.6.9 (no Linux 2.4 is available for Itanium) and does not provide any implementation variants.

4 Results

Figures 4 to 6 visualize the measured execution times for kernel entry and exit using different system configurations as discussed in section 3.4. Every bar represents a certain configuration and consists of six sections. Figure 3 illustrates which level of gray corresponds to which section of the entry and exit paths in the timing figures.

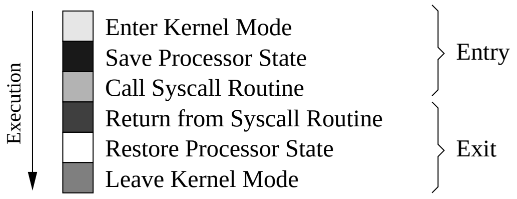

FIGURE 3: *Measured Entry and Exit Sections*

Of course, the order of the individual sections is always the same. After each bar, the total time for the combined entry and exit path execution is given in microseconds. For visual comparison, figures 4 to 6 are scaled identically.

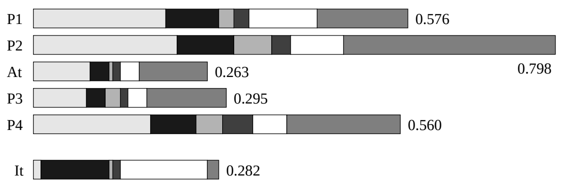

FIGURE 4: *Standard Interrupt Entry/Exit, different Architectures*

The first question to answer was how much time the conventional interrupt-based Linux entry and exit actually needs. Figure 4 illustrates the results. It is interesting to note that the Pentium 4 is almost as slow as the original Pentium MMX, whereas Athlon and Pentium III need only approximately half of that time. It can be seen that not only the privilege switch is faster for the latter systems, but also the inner sections. The Itanium is almost equally fast, its time for privilege level changes is especially low.

The second question we investigated was a comparison between the traditional interrupt-based entry/exit and the more recent sysenter/sysexit mechanism. Figure 5 depicts the respective times for the architectures providing sysenter/sysexit. Again, the Athlon and Pentium III are in the lead. Pentium II and Pentium 4 need more than twice the execution time.

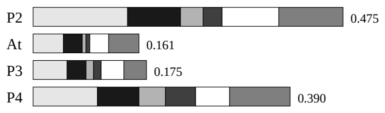

FIGURE 5: *Sysenter/Sysexit, different Architectures*

Comparing figures 4 and 5 we can conclude that on every analyzed architecture sysenter/sysexit is more efficient than interrupts . The P4 is faster by a factor of 1.4 and all other architecture even by a factor of 1.6.

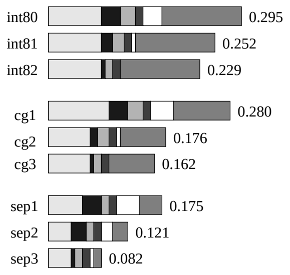

FIGURE 6: *Pentium III, different Implementation Variants*

The last aspect discussed here is how the implementation variants perform on one and the same architecture. As an example, figure 6 shows the obtained timing parameters for the Pentium III. The call gate mechanism needs approximately the same time as the interrupt (this is in fact true for all architectures). Sysenter/sysexit is considerably more efficient. The variants with reduced functionality (int82, cg3, sep3) represent the performance of the pure mechanism without any operating system overhead. It can be seen that the fastest mechanism needs no more than 82 nanoseconds for a combined entry and exit.

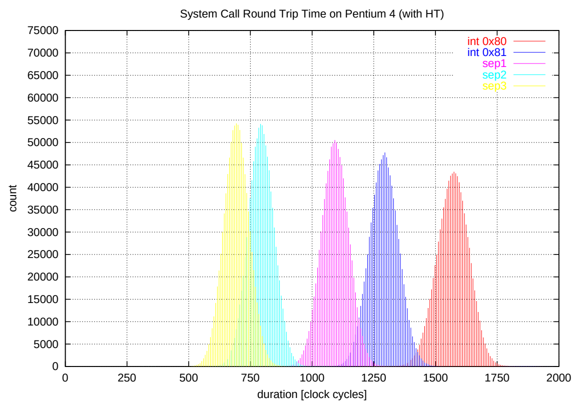

FIGURE 7: *Histogram of Entry/Exit Timing for Pentium 4*

When obtaining measurement times for the Pentium 4 we noticed an interesting phenomenon. Regardless of the implementation variant, the individual execution times varied to a considerable degree. Figure 7 shows histograms of the execution times of some variants of entry/exit paths. The bell-like shape and size of the histograms are almost identical. Furthermore, the individual execution times differ exactly by seven clock cycles! The explanation of that unusual behavior is beyond the scope of that paper, though. All other architectures (including the Intel Itanium) exhibited almost constant execution times.

5 Conclusions and Outlook

We presented a methodology to cycle-accurate measure operations crossing the user-kernel boundary. Special attention was payed to minimize timing influences by the measurement process itself.

We proved that sysenter/sysexit is the most efficient syscall mechanism regardless of the operating system overhead and quantified its performance advantage over the traditional mechanisms. Second, we observed that system path efficiency does not simply follow processor evolution. Apart from the Pentium 4 all processor architectures exhibited a very predictable timing. The reason for the Pentium 4's timing variations is a subject of further research.

To broaden the performance comparison we encourage interested readers to port our methodology to architectures and kernel versions not analyzed so far. The kernel patch and the test application can be downloaded at *http://rtg.informatik.tu-chemnitz.de* .

References

[1] John Levon, 2002, *Profiling in Linux HOWTO*, Revision 0.1

[2] Opersys, Inc., *Linux Tracing Toolkit*, `http://www.opersys.com/LTT/`

[3] Yaoping Ruan, Vivek Pai, 2004, *Making the "Box" Transparent: System Call Performance as a First-class Result*, Proc. USENIX 2004, pp. 1–14

[4] Sven Schneider, 2005, *Profiling and Comparison of Operating System Entry and Exit Paths*, Student Research Project, Chemnitz University of Technology

[5] Ariel Tamches, Barton P. Miller, 1999, *Fine-grained dynamic instrumentation of commodity operating system kernels*, Proc. OSDI, New Orleans, pp. 117–130