Abstract—Operating systems are one of the most complex kinds of software systems ever built. Their complexity results from many factors, in special, the huge size and low-level issues. As consequence, there are many programming challenges to be considered at either the in-the-large level or in-the-small level. Program comprehension is a crucial problem for developers who need to contribute or reuse the code base in some way. One of the most important challenges in this scenario is to find the functions in the source code that are responsible for a specific feature of the system. Previous work has shown the benefits of using execution trace information to improve the comprehension process of features that can have their execution adequately reproduced. However, the use of execution traces in the comprehension of operating system kernel features is poorly studied. In this work, we show that execution traces can effectively improve program comprehension of kernel features when adequate filters are provided to the instrumentation tools.

Keywords – program understanding, execution traces, operating systems

I. INTRODUCTION

Linux is a robust operating system that can be used either in desktops or in corporate servers of large companies, supporting several plataforms (ARM, x86, MIPS, SPARC, etc). Statistics presented in [1], [2] show that the kernel source code is under active development, and consequently the evolution of the base code comes together with this development: a new release is created each 60-80 days. Developers worldwide participate in the development of the kernel. For instance, the release 2.6.35 accounted the collaboration of approximately 1145 developers, including the participation of large companies, such as, RedHat, Novell, IBM and others. Moreover, the number of lines of code also grows with the evolution of the kernel. For instance, release 2.6.10 had almost 6 million lines of code and release 2.6.27 had more than 8.5 million lines of code. In this scenario, the task of feature location [3], i.e., the task of finding source code pieces related to user point-of-view software features, is an extremely challenging task because features may not be intuitively located in the code, and even if that was the case, there are so many number of modules and functions to be navigated that the task is still challenging. This situation is even more dramatic for newbie developers working on the kernel.

In order to alleviate this challenge, the source is distributed together with a folder that contains the kernel documentation. There are still some other sources of information that explains the kernel source code [6], [7], [8], [9]. However, the available documentation is still not sufficient to reference and explain all source code files or all functions that are used to implement the feature of interest. In this way, the documentation only offers a general view of a feature, and not necessarily all features that a developer may be interested are described in the documentation. Moreover, some books had already published on the kernel [7], [9], but still there is no guarantee that the text included in the book will remain updated, considering the kernel evolution throughout the time.

Fortunately, many solutions to the program comprehension problems had already been proposed. The solutions are based on a variety of techniques: static techniques which usually construct graphs from the source code using compiling techniques and perform some kind of query or browsing on that graph; dynamic techniques which usually extract information from the execution of the desired feature to drive the location of the piece of interest; information retrieval techniques which consider the terms written by the programmers to associate with feature terms and enable queries by similarity; and hybrid techniques, which seems to be the most successful ones, that combine the above techniques to enhance the precision and recall of the returned information.

However, program comprehension using dynamic information has not been studied with operating system kernels as the target software [3], [4]. One possible reason is that, if the analysis of dynamic information from the execution still imposes important challenges for systems implemented in high-level languages, such as Java, where the events captured during the execution are typically function entry and exit, consider the situation where important events to be considered in operating system execution traces could be associated with low-level interfaces, such as, interrupt handling. Moreover, the number of different kinds of events to be considered by the instrumentation tool could also be an issue.

Despite this negative scenario for using information of execution traces in the problem of program comprehension for operating systems kernel, we raise the hypothesis that developing appropriated filters into the instrumentation tool may provide useful and feasible information for developers.

---

1 This work was partially supported by the Brazilian agency CAPES.
The goal of this work is to develop a method for producing filters for low-level events that could accurately extract dynamic information of instrumented programs, without incurring in the problem of flooding the developer with unnecessary trace information. The more filters are precise, the more developer’s work is simplified. The suitable filters should enable the use of execution trace information for program comprehension problems in operating systems, especially those related to locating pieces of code related to some specific functionality.

In the next section, the tools used in this study will be reviewed. In Section III, we present the study setting. In Section IV the results of the study are presented and in Section V these results are discussed. Finally, Section VI presents our final remarks.

II. BACKGROUND TOOLS

A. Ftrace

The ftrace is a tool used to trace kernel functions [12], [13]. It is a simple tracer of kprobes, but faster and more precise in terms of time analysis [11]. Ftrace does not require any specific application in the user-space and all configurations steps are carried out using the debugfs file system. Ftrace can be used in latency analysis or in debugging performance problems. The ftrace infrastructure allows other types of tracing, such as, irqsoff plugin, which traces source code areas that disable interrupts. This plugin allows to measure the interval times in which interrupts are disabled in the kernel. Ftrace was used in this work to collect traces of the subject program proposed in the experiment.

B. SystemTap

SystemTap provides a scripting environment to analyze the tasks of a Linux system at runtime [10]. The dynamic information can be collected at real-time, which is a flexible way to trace the kernel execution. SystemTap provides mechanisms to extract, filter and summarize in order to simplify the analysis and diagnosis of the desired properties. One positive point is that SystemTap does not require to recompile, to reinstall and to reinitialize the kernel in order to obtain the target data [11].

SystemTap uses an internal scripting language similar to the AWK programming language. An internal analyzer checks the script for syntactic errors and converts it to a C program, which is loaded as a kernel module, as shown in Figure 1. SystemTap also allows creating modules for other versions of the kernel, other than the version of its respective running kernel. It is possible to copy the module to the target system and execute it with the program staprun.

The primary use of SystemTap in this work is to trace Linux kernel events occurring during a per-specified period of time. SystemTap will be used to find the mean time used by the clock tick handler and also to find how many clock ticks had occurred during the execution of a subject program in the proposed experiment to validate the quality of the experiment result.

C. Hardware interface

In this work, the target architecture will be based on SMP (Symmetric multiprocessing) used in several models of Intel x86 processors. In this architecture, the APIC system (Advanced Programmable Interrupt Controller) is responsible for generating and managing several kinds of hardware interrupts [15]. Figure 2 shows a typical APIC system, basically consisting of an I/O APIC module responsible to receive interrupt requests from I/O devices (keyboard, network, disks, etc.) and forward them to the Local APIC module (LAPIC), which is integrated into the processor. Each LAPIC contains a set of APIC registers and associated hardware that control the feeding of interrupts to the processor core.

The main source of interrupts are: 1) I/O devices locally connected, i.e., interrupts generated by I/O devices connected directly to the connectors of local interrupts of the processor (LINT0 and LINT1); 2) I/O devices connected externally, i.e., interrupts generated by I/O devices connected to the module I/O APIC; 3) inter-processor interrupt (IPIs), i.e., a processor can use the IPI mechanism to interrupt other processor or a group of processors; 4) interrupts generated by the timer APIC, i.e., the local APIC timer that can be programmed to send periodical interrupts i.e., *clock ticks*; 5) interrupts generated by the temperature sensor; 6) internal error APIC interrupt signaled when an internal error of the APIC is detected.

In this work, the studied problem will require disabling and enabling timer interrupts (4), to avoid that tasks do not be periodically interrupted.

![SystemTap operation.](image)
III. THE STUDY SETTING

In this section, we will present the experiment designed to evaluate the feasibility of using execution traces for program comprehension of operating system kernels.

The experiment design was defined using several components:
1. A problem that required some program comprehension activity in the target operating system kernel;
2. The solution/contribution to the problem cited in the previous item;
3. The method used to support and evaluate the construction of the solution cited in the previous item.

The problem used in the design of the experiment of this study was the OS Jitter that is one of the main factors that can introduce delays when processing applications in high performance computing environments – HPC. OS Jitter can be considered any interference that an application process or thread experiences because of the execution of internal tasks the operating system. In this study, the Linux kernel was chosen because of the availability of its source code and because it is being used in 91.4% of top 500 supercomputer systems [14]. Among the several types of interferences of the operating system during user application execution, especially for HPC applications, we can highlight: the execution of administrative process (daemons) and periodical kernel routines, such as, the clock tick.

In this study, the chosen contribution is to understand where OS jitter caused by clock ticks is implemented in the kernel. The proposed solution to reduce OS jitter is based on tickless processors, where the CPU is assigned to user application and it will not be interrupted by periodic kernel tasks (clock ticks).

Currently, the Linux kernel already supports a similar approach – tickless kernel – to reduce energy consumption. So, the maintenance task is to reuse the current implementation of tickless kernel of Linux to introduce the feature of executing CPU-bound process without clock interrupts. In this study, the user application that will be executed as a CPU-bound process is a matrix multiplication program.

The method used to support and evaluate the quality of the comprehension process of where clock ticks are handled in the Linux kernel consists of: the general strategy to filter execution traces to find the desired feature in the source code and the validation of the filtering process.

A. Trace Filtering

The filtering strategy needs to isolate the execution of user application (matrix multiplication) to only one CPU, in such way that the events captured in traces are only related to the respective user process. The user program must be instrumented in the following way:

1. Configure the system to run the process only in CPU 1 (sched_setaffinity system call).
2. Turn off the trace capture of ftrace; configure ftrace to capture the trace of CPU 1 only; cleans the log file of ftrace.
3. Configure the system to run all other tasks, threads, daemons in CPU 0.
4. Move all timers – scheduled tasks (workqueues, callouts, etc.) from CPU 1 to CPU 0.
5. Move all interrupts (network, keyboard, mouse, etc) to CPU 0, except timer interrupt.
6. Activate ftrace trace collector.
7. Capture the initial time (clock_gettime).
8. Allocate dynamically a matrix 1024x1024.
9. Perform the core application (matrix multiplication).
10. Capture the final time (clock_gettime).
11. Deactivate ftrace trace collector.
12. Get the time used in the user application.

This strategy should guarantee that the trace collected by ftrace contains events (function calls) related to the user application and to the timer interrupt handler.

It is expected that function(s) related to timer interrupt handler should be recognized in a reasonable way using the generated trace.

B. Validation of the result of the filter output

In order to validate the quality of the generated trace, the instrumented user program described previously in the Trace Filtering subsection was modified to include a system call between steps 5 and 6 to disable clock tick handling.

Figure 3 shows the system call that should be used with parameter 1 to disable clock ticks. Figures 4 and 5 contain the code to effectively perform the disabling process.

```
1 int sys_confapic(struct pt_regs *regs) { //ID=337
2   int pparameter = regs->bx;
3   if (pparameter == 1)
4     lapic_suspend(NULL);
5   if (pparameter == 2)
6     lapic_resume(NULL);
7 }  
```

Figure 3. Function sys_confapic
Figure 4. Lapic suspend

```c
void disable_local_APIC(void) {  
  unsigned int value;  
  if (x2apic_mode & !lapic_phys)  return;  
  clear_local_APIC();  
  value = apic_read(APIC_SPIV);  
  value &= ~APIC_SPIV_APIC_ENABLED;  
  apic_write(APIC_SPIV, value);  
  #ifdef CONFIG_X86_32  
  if (enabled_via_apicbase)  
    unsigned int l, h;  
    rdmrs(MSR_IA32_APICBASE, l, h);  
    l &= ~MSR_IA32_APICBASE_ENABLE;  
    wrmsr(MSR_IA32_APICBASE, l, h);  
  #endif  
  if (intr_remapping_enabled)  
    disable_intr_remapping();  
  if (intr_remapping_enabled)  
    int flags;  
    l &= ~MSR_IA32_APICBASE_ENABLE;  
    wrmsr(MSR_IA32_APICBASE, l, h);  
}  

Figure 5. Disable Local APIC

It is also necessary to include a system call between steps 10 and 11 to enable clock interrupts again on the chosen CPU; otherwise it would become inaccessible to other processes after the execution of the experiment. Figure 6 shows the necessary code to re-enable clock ticks.

Figure 6. Fragment of lapic_resume

The resulted trace after running the instrumented system with clock tick disabling should be compared with the trace where clock ticks were not disabled, in order to see if the filtering process that selected the considered function(s) to implement clock tick handling was correct. In order to confirm the correction, that function(s) should not be present in the new trace.

Another adopted validation step was to verifying the number of calls to the function(s) that are supposed to be the one(s) that handle clock ticks using the SystemTap script shown in Figure 7. A complementary SystemTap script shown in Figure 8 calculates the duration of the execution of these respective functions that can indicate some measure of the OS jitter.

Figure 7. Script to verify the number of clock tick handling

```c
static int lapic_resume(struct sys_device *dev) {  
  unsigned int flags;  
  int maxlvt;  
  if (!apic_pm_state.active)  
    return 0;  
  maxlvt = lapic_get_maxlvt();  
  ...  
  local_irq_restore(flags);  
  if (intr_remapping_enabled)  
    close_local_APIC();  
  if (!x2apic_mode)  
    enable_x2apic();  
  else {  
    rdmsr(MSR_IA32_APICBASE, l, h);  
    l &= ~MSR_IA32_APICBASE_ENABLE;  
    wrmsr(MSR_IA32_APICBASE, l, h);  
    if (maxlvt >= 4)  
      apic_pm_state.apic_lvtt = apic_read(APIC_LVTT);  
    apic_pm_state.apic_lvt0 = apic_read(APIC_LVT0);  
    maxlvt = lapic_get_maxlvt();  
    ...  
    enable_x2apic();  
  }  
  free_ioapic_entries(ioapic_entries);  
  WARN(1, "Saving IO-APIC state failed: %d", ret);  
  free_ioapic_entries(ioapic_entries);  
  goto restore;  
  mask_IO_APIC_setup(ioapic_entries);  
  mask_8259A();  
}  

```
global i;
global j;
global var1;

probe begin { i=0; }
probe kernel_function("<name of the supposed function>") { call {
    if (cpu()== $1) { var1[$1]=gettimeofday_ns() }
}
probe kernel_function("<name of the supposed function>") { return
    if (cpu()== $1) {
        j=gettimeofday_ns()
    }
    i++
    if (i==310) exit()
}
}

Figure 8. Script to analyze the duration of clock tick handling

IV. RESULTS

This section presents the results obtained after running the method proposed in the previous section.

A. Trace Filtering Result

The result of trace filtering is shown in Figure 9. The trace consists of function calls with their respective nesting, which can be considered as a call tree. Approximately, 90% of the trace consists of calls to `smp_apic_timer_interrupt` and their corresponding nested calls. A straightforward analysis of this particular function has shown that this is the function that should be responsible for handling clock ticks.

```c
smp_apic_timer_interrupt() {
    native_apic_mem_write();
    irq_enter();
    idle_cpu();
    hrtimer_interrupt() {
        ktime_get();
        __run_hrtimer();
        __remove_hrtimer();
        tick_sched_timer();
        ktime_get();
        tick_do_update_jiffies64();
        __spin_lock();
        do_timer();
        update_wall_time();
        update_xtime_cache();
        calc_global_load();
    }
    ...
}
```

Figure 9. Trace fragment with clock tick handling

B. Validation Results

The execution of the instrumented user program disabling clock tick handling before the core program (matrix multiplication) produced the trace shown in Figure 9. This trace is much smaller than the one shown in Figure 8. It also could be observed that there was no call to `smp_apic_timer_interrupt`.

Using the scripts shown in Figure 7 and 8 in the instrumented program that handle clock ticks, we encountered that the time to handle each clock tick is approximately 5652 nanoseconds (mean time from 310 replications) and the number of clock ticks that occurs during a 5 minutes interval is approximately 32234 (mean number from 35 replications).

Using the scripts shown in Figure 7 in the instrumented program that do not handle clock ticks, we encountered that still 4 clock ticks were handled. Indeed, 4 is a very small number compared to 32234 that would not invalidate the interpretation that the `smp_apic_timer_interrupt` is indeed the function responsible for clock tick handling. These 4 clock tick handling occurrences may have occurred during the time necessary to initiate the script and the user program execution which are not exactly simultaneous.

V. DISCUSSION

Other approaches to analyze execution traces have been extensively used in the program comprehension [4, 16]. However, there is no study whose target system is an operating system kernel. The seminal work in the application of execution trace to program comprehension is the Software Reconnaissance approach that also compares code executed in traces with and without the features [5]. This is an interesting approach but depending on the size of trace, the excess of information may hinder the approach with useless information. In [17], the authors proposed an enhancement in the trace differentiation providing more contextualized differentiation with trace alignment algorithm and also had to propose a trace summarization algorithm in order to dramatically reduce the size of trace, and consequently produce a feasible approach. In this work, we had the same challenge of reducing effectively the size of trace in order to grasp adequately the trace file to extract the desired information of clock tick handling. We proved that our approach was effective because the desired function was readily found. It is important to note that if a newbie kernel developer had to look for this function in a traditional way, browsing the source code from the `main`
function until he could find the method, this would be a very hard task that could take several days or even weeks of work.

In [18], another approach to summarize traces was designed using fan-in and fan-out metrics, which is completely different from the specialized filtering approach used in this work. In their approach, the functions are ranked considering their relative importance based on graph metrics which do not consider any semantic information of the respective function. The approach of program comprehension presented in this work guides the developer in a much more precise manner, because it is the developer itself that have the tacit knowledge of what he really needs and thus, he writes the proper filters considering this knowledge. Consequently, this strategy improved dramatically the quality of the information available in the trace. However, it is important to note that our approach requires a more specialized preparation of the trace filtering. Although, the developer can reuse the generic parts of our filtering framework, he stills needs to grasp a filtering solution that will provide a precise “fishing” of the desired functions.

Some authors [19, 20] suggest the integrated use of static and dynamic views of the software system in program comprehension activities. The dynamic views are obtained by profiling the most used features in the system. Nonetheless, the primary goal is to obtain the system architecture to reduce the effort of comprehension of the system. In our approach, we have not focused on the most used features for comprehending the system in an overall manner. We have focused on a well chosen feature in order to provide direct information customized for the process of program comprehension. This requires less effort to make further needed changes in the source code. Nonetheless, we could still improve our process of writing adequate filters and even of browsing the resulted traces with static information. For instance, the use of information retrieval techniques could also be incorporated [21, 22, 23], because we could see that the function names used in kernel are very representative and similar to the developer terms in high-level communication.

VI. Final Remarks

This paper has presented an innovative study in the program comprehension area because it used execution trace information to isolate desired functions in the Linux operating system kernel. Previous studies in this area did not handled OS kernels.

Our findings have shown that despite the huge amount of events that an instrumentation tool can generate, especially in the case of OS kernel which has to cope with much more kinds of different low-level events than a high-level application, the use of proper filtering mechanisms in the instrumentation tool can reduce dramatically the complexity of the execution trace information. This scenario reduces the developer effort during maintenance tasks when he needs to find specific functions in the source code. Future work includes reproducing the subjacent methodology designed in this work in a large scale experiment to produce stronger evidences on our findings. Also, introducing hybrid static techniques in a cohesive methodology is an important step.

REFERENCES