Virtual Timers: Using Hardware Physical Timers for Profiling Kernel Code-Paths

Dimitrios Xinidis[†], Michail D. Flouris, and Angelos Bilas[†]
Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)
P.O. Box 1385, Heraklion, GR 71110, Greece
Email: {dxinid, flouris, bilas}@ics.forth.gr

***Abstract*—**

When evaluating the performance of commercial workloads it is important to be able to examine overheads induced by the operating system kernel. Currently the only method for understanding kernel overheads is by using sample-based profiling tools. Although profiling can be very useful, one of its main limitations is that it does not allow the precise profiling of specific code-paths. Using hardware timers with a simple API allows addressing this issue, however, does not deal with context switch operations that incur during code execution, due to synchronous or asynchronous events.

In this work we propose a new interface for virtualizing physical timers. Our interface takes into account context switches and provides the system programmer with simple calls to profile specific code-paths. We implement this API in the Linux kernel and demonstrate how it can be used to profile the system overhead of MySQL running a subset of TPC-H over iSCSI. Using our virtual timers we identify the time is spent in I/O related code-paths in the kernel with little effort. We also examine the overhead of the instrumentation code and find that it is less than 3% of the execution time in all experiments we perform.

I. Introduction

Profiling kernel overheads for commercial applications is an important part of understanding overall system bottlenecks. For instance, most application I/O operations currently pass through the kernel, since I/O protocols stacks are almost entirely implemented in the kernel.

[†]**Also, with the Department of Computer Science, University of Crete, P.O. Box 2208, Heraklion, GR 71409, Greece.**

Current tools for profiling kernel code are mostly based on sampling techniques [3], [10], [2]. Sampling-based profilers, periodically sample the program counter of the system and use available symbol information to associate each sample with a specific function in the executable file. Then, the total execution time is divided to the various kernel functions based on the number of samples that were accounted to each function during the sampling phase. Sampling profilers can thus operate transparently, without requiring system or application code instrumentation, but only access to symbol information. However, sampling profilers have also important limitations that make them unsuitable for profiling certain aspects of kernel overheads.

First, with a sampling profiler it is impossible to distinguish in which code-path a specific function is called. For instance, if a specific function is called in several code-paths (i.e. sequences of function calls), the time spent in this function will appear as a single statistic in the profiling results. Thus with sampling profiling we are not able to attribute this overhead to different code-paths that happen to call this function. Furthermore, it is sometimes important to measure the latency of specific operations. Sampling profilers can only provide approximations to latency calculations by combining them with a frequency counter that allows conversion of overall statistics reported by the profiler to per-instance averages. However, in many cases higher precision measurements are necessary, especially when profiling low latency operations in the kernel. Finally, when symbol information is not available for parts of

```
#samples  %     app name       symbol name
#
183251 50.1147 mysqld         (no symbols)
28151   7.6986 vmlinux        file_read_actor
19084   5.2190 vmlinux csum_partial_copy_generic
14651   4.0067 dl2k.o         rio_interrupt
11955   3.2694 vmlinux  __generic_copy_to_user
11849   3.2404 vmlinux        csum_partial
6738    1.8427 dl2k.o         start_xmit
6730    1.8405 vmlinux        schedule
4860    1.3291 vmlinux        eth_type_trans
4627    1.2654 vmlinux        kmem_cache_free_one
3091    0.8453 dl2k.o         receive_packet
2230    0.6099 vmlinux        UTM_STOP
1914    0.5234 vmlinux        tcp_v4_rcv
1821    0.4980 vmlinux        tcp_rcv_established
1764    0.4824 vmlinux        UTM_START
1589    0.4346 vmlinux        __kmem_cache_alloc
1500    0.4102 vmlinux        IRQ0x11_interrupt
1421    0.3886 vmlinux        tcp_transmit_skb
1384    0.3785 vmlinux        tcp_recvmsg
1345    0.3678 vmlinux        stop_active_timers
1327    0.3629 vmlinux        start_active_timers
1211    0.3312 vmlinux        kfree
1203    0.3290 vmlinux        default_idle
1161    0.3175 vmlinux        kfree_skbmem
1061    0.2902 vmlinux        __make_request
1029    0.2814 intel_iscsi.o iscsi_sock_msg
972     0.2658 vmlinux       tcp_v4_checksum_init
841     0.2300 vmlinux        alloc_skb
835     0.2284 vmlinux        rmqueue
817     0.2234 vmlinux       do_generic_file_read
794     0.2171 intel_iscsi.o param_equiv
```

Fig. 1. Example of Oprofile symbol sampling output

the code, interpreting profiling results may either be complicated or even impossible.

As a specific example, consider profiling MySQL [12] for kernel storage I/O overheads, e.g. using a networked storage subsystem over iSCSI [9]. A sampling profiler provides a detailed list of sampled kernel symbols, showing how much time the kernel spends in each symbol, as shown in Figure 1. However, these samples cannot be used to construct a hierarchical breakdown for different kernel layers, because there are many common functions used throughout the kernel. For example, `page_alloc()` is used to allocate a memory page. This function is used in various kernel components, including the file system, the SCSI and iSCSI layers, the TCP/IP stack, and various drivers, such as NIC and disk drivers. Knowing that the kernel spends a specific amount of time in `page_alloc()` does not allow us to provide a detailed breakdown of the kernel time in the various layers of the I/O protocol stack.

To address this issue, we examine how traditional hardware timers can be used to profile specific code-paths in the kernel. Conceptually, the required operations are fairly simple. We need the ability to start and stop a high-accuracy timer at specific points during code execution. The main problem with using stop-watch timers arises from the fact that many code-paths in the kernel include waiting on various events. If a timer has been started and the execution includes a wait on some system event that will result in a context switch, e.g. receiving a packet or completing an I/O request, then the resulting measurement may not be precise. The high frequency of event waits, context switches, and asynchronous interrupts when executing kernel code, limits the usefulness of simple stop-watch timers.

To address these issues, we examine and propose extensions to physical timer interfaces that provide stop-watch virtual timers and simplify profiling of kernel code-paths. We clarify the various possibilities and implement the proposed API in Linux. We deal with synchronous and asynchronous scheduler and interrupt events transparently to tasks. Thus, the programmer is only required to insert appropriate start/stop calls at the boundaries of the code that needs to be profiled, without worrying about events that occur in between.

To demonstrate the effectiveness of our approach we show how it can be used to profile the system overhead of MySQL running TPC-H over iSCSI. Compared to traditional physical timers, our approach requires an order of magnitude less effort. Finally, we measure the execution time overhead of the Virtual Timers and find that it is less than 3% of the execution time in all experiments we perform.

The rest of the paper is organized as follows. Section II discusses the problem and the proposed timer API. Section III discusses our Linux implementation providing all necessary background about Linux scheduling and interrupts. Section IV demonstrates how our approach simplifies profiling of kernel code-paths and discusses the instrumentation overheads it incurs. Finally, Section VI draws our conclusions.

```
inline long long GetCurrentCycles(void) {
  LARGE_INTEGER val;

  __asm__ __volatile__("rdtsc ":
       "=a" (val.split.LowPart),
       "=d"(val.split.HighPart));

  return val.QuadPart ;
}
```

Fig. 2. Function for reading cycle counter in the x86 architecture.

II. Timer Semantics and API

Most CPUs today provide cycle counters that can be used for profiling purposes. For instance the x86 architecture provides a 64-bit cycle counter that can usually be read with a single assembly instruction. Figure 2 shows how this is achieved in the x86 architecture, by inlining assembly in a C function (this is gcc-compatible code). This assembly instruction reads the value of a processor register that counts clock cycles since the last boot of the system. Clock cycles are a very accurate measure of time for most purposes. Moreover, this profiling method incurs very low overhead, requiring a single assembly instruction to read the cycle counter. In contrast, using OS facilities such as `do_gettimeofday()` that use the system real-time clock have two main disadvantages: (i) They need several instructions to compute the time and thus, incur high overhead. (ii) The granularity they provide is in the order of milliseconds, while cycle counters can potentially measure nanosecond intervals on modern high-frequency processors.

A generic stop-watch timer has essentially five primitive operations: `alloc/dealloc()`, `start()`, `stop()`, `clear()` and `read_time()`. `alloc()` creates a new timer. `start()` starts counting time by reading the current timer value and storing it. `stop()` reads the current timer value, subtracts from it the previously-stored timer value and stores their difference. This difference indicates the time elapsed between the start and stop points. The timer can then be restarted and stopped, accumulating the total time for all intervals. `read_time()` returns the current total, i.e. the elapsed time. `clear()` resets the total time measured by the timer. Finally, when the timer is not needed, we can free it using `dealloc()`.

```
int alloc (_timer_ctx_t *tc,char *name,char flag);
int dealloc (int timer_id, _timer_ctx_t *tc);
void start (int timer_id, _timer_ctx_t *tc);
void stop (int timer_id, _timer_ctx_t *tc);
void clear (int timer_id, _timer_ctx_t *tc);
```

Fig. 3. Virtual timers API.

Using the above timer concepts, profiling a code-path requires merely including the path in a pair of `start_timer()`, `stop_timer()` calls. Using multiple timers, allows us to measure independent code-paths.

This simple profiling approach, however, becomes more complicated when applied to modern operating systems, mainly because of two features: (a) multi-tasking and preemption and (b) interrupts.

Modern operating systems try to minimize blocking and wait time by overlapping execution of concurrent kernel and user tasks as much as possible. For this reason they switch between tasks whenever a task needs to wait on an event. Processes are placed in various event queues and the operating system scheduler selects the next process to run on the CPU. Since stop-watch timers use physical time between the start and stop operations, any measurement of elapsed time will include all time between these two points, i.e. the time other processes may have run on the CPU. A similar situation occurs with interrupts (software or hardware) that are issued during the run time of a process. In this case the interrupt handler routine will be included in the time of the specific code-path.

To deal with these issues in a transparent manner, our framework offers two kinds of timers: *physical* and *virtual*.

- *Physical* timers that measure all time between start and stop operations.
- *Virtual* Virtual timers automatically remove all wait and interrupt times.

Furthermore, each type of timer can be categorized as private or global:

- *Global* timers are visible from all tasks. Such timers can be started and stopped by any task running in the system.
- *Private* timers are visible only inside the context of a single task and cannot be accessed by other tasks.

Thus we can have the following combinations of timer semantics:

- *Physical-Global*: These timers are the simplest timers, traditionally available for profiling.
- *Physical-Private*: These timers are useful when a task needs to profile other activity, e.g. wait time that occurs during its execution.
- *Virtual-Private*: This type of timer can be used to profile paths in a single task without interfering with other system or user tasks.
- *Virtual-Global*: These timers do not provide useful semantics, since virtual timers make sense only in the context of a single task.

Out of the three combinations of timer semantics, the most challenging to provide is virtual-private. Supporting virtual-private timer semantics requires two additional internal functions that are not part of the traditional stop-watch timer API: `pseudo_start()`, `pseudo_stop()`. These functions are called either from the scheduler or from interrupt handlers in order to start/stop active timers of a task during a scheduling or interrupt event. Thus, these are not calls that would be used when profiling code, but are rather internal calls used in the operating system.

III. Timer Implementation

In this section we describe our implementation of virtual timers for Linux.

First, we describe the Linux scheduler, for kernel versions up to 2.4.x [7]. Linux uses separate contexts for entities that can be scheduled independently. Such entities are called *tasks*. A task may be a user process, a kernel thread, or a signal. Interrupts are distinguished from tasks and run in their own context. All user tasks are preemptible. The scheduler may suspend the execution of the current user process at any time and select another process to run, according to a scheduling algorithm. On the other hand, all kernel tasks are non-preemptible (in the 2.6.x kernel they are preemptible). However, kernel tasks may yield the CPU to improve responsiveness when waiting for I/O to complete [7].

Figure 5 shows the scheduler pseudo-code. Initially, the scheduler checks the state of the current task. If the state is set to `TASK_INTERRUPTIBLE`, the scheduler checks if there are signals that may require processing. If there are no such signals, the task is removed from the queue of runnable tasks. In both cases, the flag that indicates if the task needs to be rescheduled, is disabled. After the scheduler has applied the scheduling algorithm to all tasks in the runnable queue, it selects the most suitable task to run or the `idle` task if the queue is empty. At this point, it is guaranteed that the selected task will run. The context switch merely saves the context of the previous task and restores the context of the task to run.

To implement virtual timers we need to: (i) `pseudo_stop` all timers of the previously running task that were active when the scheduler run and (ii) `pseudo_start` the timers of the task that will run next and that were active when the process was preempted. For this we need a bitmap per task indicating the task's active timers. The bits in this bitmap are set when the timer is started and are reset when the timer is stopped by the user in the task code.

Next we describe how we deal with interrupts. When an interrupt occurs the task that holds the CPU is suspended and a general interrupt handling routine is called [1], as shown in Figure 4. After the completion of the interrupt there are two possibilities: (i) If the task that was interrupted is a user task, the scheduler is called and selects a task to run. (ii) If the task is a kernel task, the control of the CPU returns to this task. Consider, for example, that a task T1 uses the CPU. When an interrupt arrives, the OS saves the context of T1 in the top-most interrupt handler and calls the `do_IRQ` function, which is the entry point of all interrupts. `do_IRQ` calls other functions, including the interrupt handler of the specific interrupt. After the interrupt is handled, the OS decides which task should be scheduled next, based on the reschedule flag of the interrupted task. To deal with interrupts we instrument the generic interrupt handler of the Linux kernel. Similarly to the scheduler, in the beginning of this function we `pseudo_stop` the active timers of the interrupted task. When the interrupt routine is completed we `pseudo_start` the active timers of the next task to run. We note that we could avoid the `pseudo_start` operation in the case where the scheduler is called, however we prefer to include it because it is easier to implement in the current Linux code.

A case that needs special handling arises with nested interrupts. In this case and during a nested interrupt, we will `pseudo_stop` the active timers of the previous

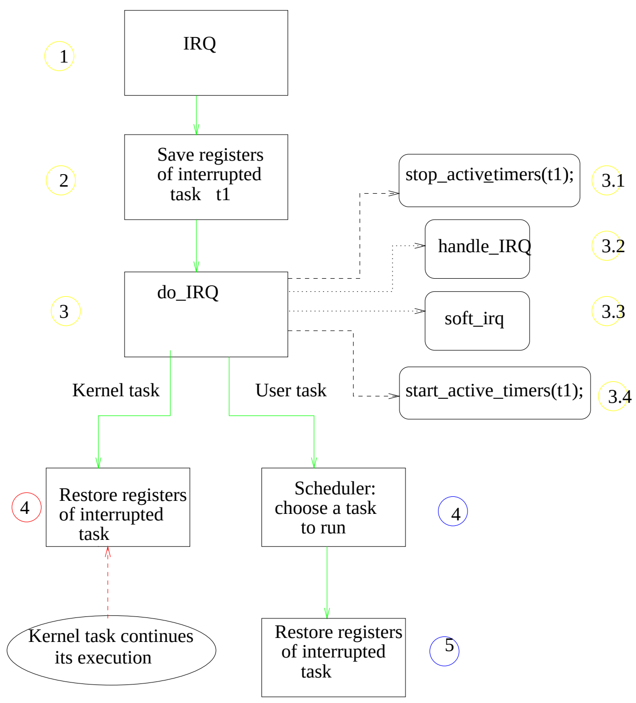

IRQ
Save registers of interrupted task t1
do_IRQ
stop_activetimers(t1);
handle_IRQ
soft_irq
start_active_timers(t1);
Kernel task
User task
Restore registers of interrupted task
Scheduler: choose a task to run
Kernel task continues its execution
Restore registers of interrupted task
1
2
3
3.1
3.2
3.3
3.4
4
4
5

Fig. 4. Generic interrupt execution path in Linux.

task. However in a nested interrupt, the previous task is the task that was originally interrupted and not the previous interrupt context. Since `pseudo_stop` accumulates the current timer difference, this will lead to multiple accumulation of a single time difference. For this reason `pseudo_stop` checks if a particular active timer has been pseudo-stopped already. Similarly `pseudo_start` checks if the timer is already pseudo-started. Thus, our implementation stops and starts virtual timers only at top-level interrupts. We measure the frequency of nested interrupts during our experiments and we find that they account for less than 4% of the total `pseudo_start`, `pseudo_stop` calls.

Finally, another scenario that requires special handling is the case where an interrupt occurs while the scheduler itself is running, and the virtual timers framework has pseudo-started the active timers of the next task to run and we are just before the actual context switch. In this case execution is still in the context of the current task but the framework has started the timers of the next task to run. As mentioned previously, the virtual timers code in the interrupt handler will stop the active timers of the task that

```
schedule() {

need_resched_back:
    prev = current;
    if ((prev->state == TASK_INTERRUPTIBLE)
               && signal_pending())
        prev->state = TASK_RUNNING;
    else
        del_from_runqueue (prev);

repeat_schedule:
    next = idle_task ();

    search_list();

 /* c  is the  goodness value. The  biggest the
    more likely  for a process to  run next. If
    it is  0 the process cannot run  on the CPU
    that has used until now. */

    if (c == 0)  goto repeat_schedule;

    if (prev == next) goto same_process;
        prepare_To_switch ();

    stop_active_timers (prev);
    start_active_timers (next);
    switch (prev,next);

same_process:
    if (current->need_resched)
        goto need_resched_back;
    return;
}
```

Fig. 5. Pseudo-code for the Linux scheduler.

was interrupted. In this case the interrupted task appears to be the task that was preempted and not the task to run next. This means that the virtual timers for the task to be scheduled next will continue to measure time, during interrupt handling. This happens because starting virtual timers and performing the context switch are not atomic operations. The results is that the next task to run will include the interrupt time. However, this will not result in, otherwise, corrupted or incorrect measurements. One way to fix this, is to make the two operations atomic, e.g. by disabling interrupts. However, since, this is an infrequent case (it has never occurred in our experiments) we prefer to keep our implementation simple and not deal with this atomicity issue.

Our virtual timers framework use two main data structures. The first is general and is used for all tasks running

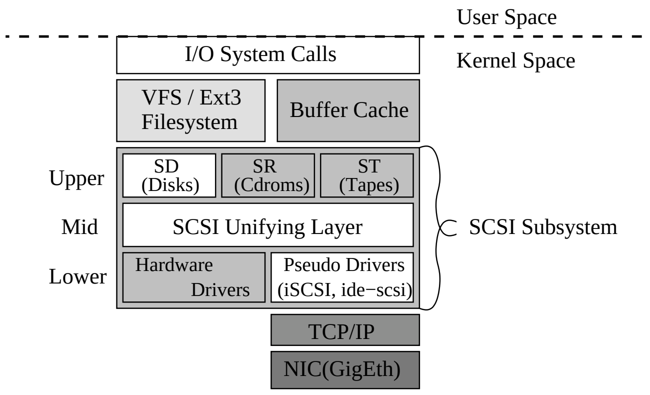


Fig. 6. Kernel layers we have instrumented with timers.

in the system. Its elements store information about the execution time of all tasks and all timers in the system. This structure is allocated at system initialization, when the idle process is created.

The second data structure is private to each task and is used to implement the virtual-private semantics. The elements and information are similar to the first, global data structure. This, second structure is attached to the kernel's `task_struct`, which contains all information for each task. The structure is allocated at the entry point of every new task in the system, `do_fork`.

Finally, all timing information is available through the `/proc` filesystem. The global statistics, i.e. all timers for all tasks, are accessible through the `/proc/ktimers` file, while separate task timings are accessed through `/proc/ktimers_"taskid"`.

IV. Results

In our experiments we use two x86 machines. Each machine is equipped with two Athlon MP2200 processors at 1.8 GHz and 512 MBytes of RAM. The nodes are connected both with a 100 MBit/s (Intel 82557/8/9 adapter) and a 1 GBit/s (D-Link DGE550T adapter) Ethernet network. All nodes are connected on a single 24-port Gigabit Ethernet switch (D-Link DGS-1024) with a 48 GBit/s backplane. The 100 MBit network is used only for management purposes. All traffic related to our storage experiments uses the GBit Ethernet network. The operating system we use is `Linux RedHat 9.0`, with the `2.4.23-pre5` kernel and our virtual timers extensions. The iSCSI implementation we use is Intel's iSCSI [6].

We evaluate our virtual timer framework by profiling the kernel I/O path for MySQL [12] running the TPC-H benchmark [11]. Figure 7 shows the *system* execution time breakdown of several TPC-H queries. The left bar in each pair is constructed using Physical-Global timers and manually subtracting the wait times in the kernel, i.e using separate timers to measure all wait times. The right bar in each pair is extracted with using Virtual Private timers to automatically remove wait times. To understand the various sections of each breakdown, Figure 6 shows the kernel layers for the iSCSI protocol path that have been instrumented. Figure 7 illustrates the time spent in each kernel layer. Time labeled as "other" is time not included in the layer breakdown we are interested in. "Other" time includes instrumentation overhead as well. More details on the results of the iSCSI performance and breakdown can be found in [13].

In our experience, manually instrumenting the kernel code to remove wait times and asynchronous events is both a tedious and error-prone process. Collecting the profiling data with Physical-Global timer semantics required several months and motivated the design and development of the Virtual-Private timer semantics. With Virtual-Private timers, we only need to insert start/stop calls at the boundaries of each kernel layer shown in Figure 7, which are fairly easy to identify.

Instrumenting kernel code with our framework incurs additional overhead in the system. To evaluate this overhead we measure the time each operation takes. Table I shows the cost for the four main API functions. The results show that the overhead of these functions is in all cases less than 3% of total execution time and on average 1.28%. This allows the framework to profile the system without significantly affecting the system's behavior.

V. Related Work

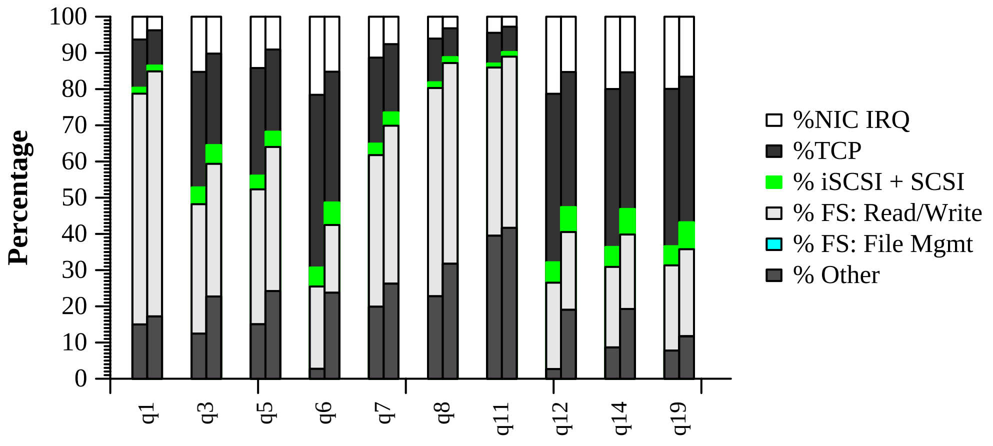
There exist several system profiling tools. We can categorize them based on the events they are able to profile:

First, system-level tools that profile operating system events, such as page faults, context switches, network traffic etc. Such tools usually profile course grain events. The Microsoft Windows System Monitor [10] counts and logs OS events and hardware resources. Similar statistics are

TABLE I

INSTRUMENTATION OVERHEAD. THE FIRST COLUMN SHOWS THE TIME IN CPU CLOCK TICKS FOR A SINGLE OPERATION. THE REST OF THE COLUMNS SHOW CUMULATIVE TIMES (SECONDS) AS WELL AS INSTRUMENTATION OVERHEAD FOR EACH QUERY. PERCENTAGES REFER TO TOTAL EXECUTION TIME.

Operation	Cost (cycles)	q1	q3	q5	q6	q7	q8	q11	q12	q14	q19
Total Execution Time	N/A	85.25	68.27	75.53	27.02	71.32	151.77	29.51	32.16	34.98	34.12
Total System Time	N/A	35.36	16.05	16.13	9.52	19.34	42.68	10.07	10.54	10.74	10.19
Total User Time	N/A	43.61	37.27	52.66	8.71	39.77	98.17	15.8	10.45	9.1	14.38
Total Idle Time	N/A	6.28	14.95	6.74	8.79	12.21	10.92	3.64	11.17	15.14	9.55
Profiling overhead	N/A	2.36%	1.03%	1.0%	0.8%	2.0%	1.5%	2.71%	0.7%	0.75%	0.8%
`start()`	560	1.06%	0.33%	0.37%	0.21%	0.50%	0.71%	1.26%	0.19%	0.19%	0.19%
`stop()`	1140	1.18%	0.56%	0.51%	0.37%	0.60%	0.80%	1.39%	0.35%	0.34%	0.34%
`pseudo_start()`	485	0.03%	0.04%	0.03%	0.08%	0.04%	0.01%	0.01%	0.07%	0.06%	0.06%
`pseudo_stop()`	1400	0.06%	0.009%	0.07%	0.17%	0.85%	0.04%	0.03%	0.15%	0.14%	0.14%



(a) Breakdown.

Fig. 7. TPC-H breakdowns. Left bars show breakdown using global-continuous timers with manually removed wait times. Right bars show breakdown with local-interruptible timers.

exported through the /proc filesystem in Linux and are displayed by utilities such as “top” [8]. Such monitoring tools are intended for system administration and not kernel development, which requires kernel code profiling information.

Second, code-level tools that profile user or kernel code. Today, such tools almost exclusively rely in some type of cycle counter to provide fine-grain measurements: Our work belongs in this second category. Most of the tools used in this category are currently based on sampling methods. Oprofile [3] is a tool that enables OS profiling, using the symbol sampling techniques to get statistics for the various symbols that are executed. Oprofile periodically, and usually at short intervals generates an interrupt that samples the value of the PC (program counter) register in the CPU. Using the PC value and the code symbol information, oprofile determines in which function the CPU was executing at that time. At the end of the experiment it reports a breakdown of the percentage of execution time that was spent in each function, based on this method. VTune [2] is a sampling profiler for x86 systems, both for Linux and Microsoft Windows. VTune also reports other CPU events by means of the x86 hardware event counters.

As discussed in Section I, these sampling profilers can neither provide a detailed breakdown of execution path nor report the exact time of execution of a kernel function, since they use approximations based on frequency counters to get the samples.

A user-level profiler that uses processor counters is PAPI [4] using the Perfctr [5] Linux kernel patch. PAPI has some similar concepts to the Virtual Timers framework, but is a user-level application profiler and is designed for portability over many OSes. PAPI provides an interface to access the hardware performance counters found on most modern processors through user-level applications. These allow application developers to measure hardware or system events (e.g. cache hits/misses, TLB misses, ops per second, etc.) per process or in total for the system. Virtual Timers on the other hand focuses on kernel-level code profiling and provides primitives for high-accuracy timing of kernel code-paths.

VI. Conclusions

In this work we examine a new interface for virtualizing physical hardware cycle timers in the kernel. Virtual timers allow the programmer to instrument code-paths without having to worry about synchronous or asynchronous events that may occur during path execution. Virtual timers deal automatically with scheduler and interrupt events, excluding all related wait times. We implement a prototype of virtual timers in the Linux kernel. Using the virtual timer functionality we are able to provide a breakdown of the kernel I/O overheads for MySQL running on a system using networked storage over iSCSI. Using physical timers for extracting code-path measurements is a complex and error-prone process. Virtual timers greatly reduce the effort required. We also examine the overhead of our virtual timer instrumentation and we find that it is always within 3% of the total execution time in our experiments. Finally, our approach does not replace sampling profilers, but rather complements them. We have found that a combination of sampling-based profiling and our virtual timers is a powerful tool for detailed kernel-level profiling.

VII. Acknowledgments

We would like to thank the members of the CARV laboratory at ICS-FORTH for useful discussions. Also, we thankfully acknowledge the support of the European FP6-IST program through the SIVSS project.

References

[1] http:// www.linux.com / howtos / KernelAnalysis-HOWTO-6.shtml.

[2] Intel vtune performance analyzers. http:// www.intel.com/ software/ products/ vtune.

[3] Oprof: profiling system for linux 2.2/2.4/2.6. http:// oprofile.sourceforge.net.

[4] Papi: Performance application programming interface. http:// icl.cs.utk.edu/ papi/.

[5] Perfctr: Linux performance counters. http://user.it.uu.se/ mikpe/ linux/ perfctr/.

[6] Project: Intel iSCSI reference implementation. http://sourceforge.net/ projects/ intel-iscsi.

[7] Scheduling in unix and linux. http:// www.kernelnewbies.org/ documents / schedule /.

[8] Unix Top. http:// sourceforge.net/ projects/ unixtop.

[9] Internet Engineering Task Force (IETF). iSCSI, version 08. In *IP Storage (IPS), Internet Draft, Document: draft-ietf-ips-iscsi-08.txt*, Sept. 2001.

[10] Microsoft.com. Monitoring and Tuning System Performance in Microsoft Windows XP. http:// support.microsoft.com/ kb/823887.

[11] Transaction Processing Performance Council (TPC). *TPC BENCHMARK H, Standard Specification, Revision 2.1.0.* 777 N. First Street, Suite 600, San Jose, CA 95112-6311, USA, August 2003.

[12] Michael Widenius and David Axmark. *MySQL Reference Manual*. O'Reilly & Associates, Inc., June 2002.

[13] Dimitrios Xinidis, Michail D. Flouris, and Angelos Bilas. Performance Evaluation of Commodity iSCSI-based Storage Systems. In *13th NASA Goddard, IEEE Conference on Mass Storage Systems and Technologies (MSST2005) (to appear)*, April 2005.