INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Developing a Software Tool for Precise Kernel Measurements

Samuel Thibault — Robert D. Russell

N° ????

June 2005

Thème NUM

ISSN 0249-0803

Developing a Software Tool for Precise Kernel Measurements

Samuel Thibault* , Robert D. Russell†

Thème NUM — Systèmes numériques
Projet Runtime

Rapport technique n° ???? — June 2005 — 25 pages

Abstract: Tracing the flow of control in a running kernel is useful for determining where performance can be improved, for instance. FKT (Fast Kernel Tracing) is a set of tools to achieve such tracing. However, its recording size was limited to the size of available memory, lacking a flushing mechanism. We here describe how such flushing was added to FKT without too much perturbing the measurements, how new issues raised by long recordings were addressed, as well as the new features that became useful.

Key-words: performance evaluation, kernel probing, synchronization, sendfile, sendpage, page cache, buffer cache, operating systems

* École Normale Supérieure de Lyon
† University of New Hampshire, Computer Science Dpt

Unité de recherche INRIA Futurs
Parc Club Orsay Université, ZAC des Vignes,
4, rue Jacques Monod, 91893 ORSAY Cedex (France)
Téléphone : +33 1 72 92 59 00 — Télécopie : +33 1 72 92 59 ??

Développement d'outils logiciels pour une instrumentation précise de noyaux

Résumé : Il est utile de tracer le flux d'exécution d'un noyau, pour déterminer où l'on peut améliorer les performances par exemple. FKT (Fast Kernel Tracing) est un ensemble d'outils permettant un tel traçage. Cependant la taille de ses enregistrements était limitée à la taille de la mémoire disponible car il ne possédait pas de mécanisme écrivant sur disque au fur et à mesure. Nous décrivons comment un tel mécanisme a été ajouté à FKT sans pour autant perturber les mesures, comment les nouveaux problèmes apportés par de longs enregistrements ont été traités, ainsi que les nouvelles fonctionnalités qui se sont révélées utiles.

Mots-clés : évaluation de performances, trace de noyau, synchronisation, sendfile, sendpage, cache de pages, cache de buffers, systèmes d'exploitation

Contents

List of Figures

1 Introduction

1.1 General presentation of FKT

FKT is a fine-grained trace tool for Pentium platforms. It uses really fast[1] software probes placed at the entry and the exit of each function to be traced, an example code is given in figure 1. The given mask is ANDed with an 'active mask' to test whether to do the recording or not. Then the TSC of the Pentium processor (Time Stamp Counter, incremented at each processor cycle) is read to get a very precise time stamp, and it is recorded along with the probe code, a function entry or exit for instance, and some additional data if wanted. See probe record format in figure 2. A slot is hence at least 12 bytes long, but generally less than 24 bytes long.

The cost of a probe is really low, so that it can be inserted anywhere in the kernel. Probes have already been incorporated for system calls, traps and IRQs, as well as bunches of functions from the TCP/IP layer, the file-system layer and the generic SCSI layer (see section B.5 for details). Bunches of probes can be selectively enabled or disabled through the active mask, so that recompiling the kernel to switch between instrumenting the file system and the SCSI layer is not needed, for instance.

```
int function(int arg1, int arg2) {
    int ret = 0;
    FKT_PROBE2(FKT_FUNCTION_KEYMASK, FKT_FUNCTION_ENTRY_CODE, arg1, arg2);

    /*
     * body
     *
     * of
     *
     * function
     */

out:
    FKT_PROBE1(FKT_FUNCTION_KEYMASK, FKT_FUNCTION_EXIT_CODE, ret);
    return ret;
}
```

Figure 1: traced function example

1.2 Examples and results

A part of the record of running `find` is shown in figure 3 for instance.

The numbers are respectively the time in cycles since previous probe entry, the pid of the current process, and the probe code. The probe code is then decoded, and its arguments

[1] written in assembly language

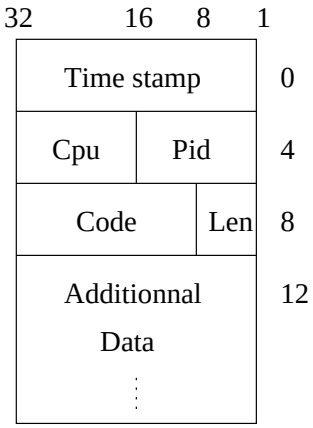

Figure 2: slot format

```
Cycles Pid Code                      Name P1, P2, P3, ...
------ --- ----                      ---- ---------------
       730 00c4            system call 196 sys_lstat64
 21975 730 0303          ll_rw_block_entry
   336 730 031c generic_make_request_entry 0
   593 730 031d       __make_request_entry
  1045 730 041d        __make_request_exit 0
   100 730 041c  generic_make_request_exit
   118 730 0403           ll_rw_block_exit
 16759 730 031a                  switch_to 0
 88845   0 022e                     IRQ 14 ide0
  5389   0 0299         end_buffer_io_sync 2, 25
  2640   0 0300          ret_from_sys_call
   513   0 031a                  switch_to 730
  3978 730 0300          ret_from_sys_call
```

Figure 3: Small part of tracing `find`

```
              Name Code   Cycles Percent
              ---- ----   ------ -------
        tcp_sendmsg 03a0  3406009   3.80% *
   tcp_transmit_skb 03a2  8610027   9.60% ****
     tcp_write_xmit 03a3  1477488   1.65%
         tcp_v4_rcv 03a9  6602416   7.36% ***
tcp_rcv_established 03ad  7237981   8.07% ***
            tcp_ack 03af  2064358   2.30% *
        tcp_recvmsg 03b0 11318084  12.62% *****
     tcp_recvmsg_ok 03b2 47725129  53.23% ************************
       tcp_sync_mss 03bd     7833   0.01%
       tcp_send_ack 03b7  1166403   1.30%
           tcp_data 03ae    37117   0.04%
       tcp_send_skb 03a1      734   0.00%
    Total elapsed cycles 89653579 100.00%
```

Figure 4: tcp statistics

```
              Name Code   Cycles Count Average Percent
              ---- ----   ------ ----- ------- -------
         tcp_v4_rcv 03a9    38497    63     611   0.06%
tcp_rcv_established 03ad   379954   267    1423   0.59%
     tcp_recvmsg_ok 03b2 47835471 10040    4764  73.76%
       tcp_send_ack 03b7  5280944   891    5927   8.14%
   tcp_recvmsg ONLY 03b0 11318084  2146    5274  17.45%

        tcp_recvmsg 03b0 64852950  2146   30220 100.00%
```

Figure 5: detail of function calls and cycles spent by function `tcp_recvmsg`

printed. This example clearly shows what happens when find encounters a file whose inode wasn't in the page cache already: after some work, a block request is queued, the idle task is then schedule, since there is nothing to do but wait for the `ide` IRQ, which puts an end to the IO, schedule back `find`, which can then return the result.

A more advanced analysis example is getting a file via TCP, shown in figure 4.

`tcp_recvmsg_ok` is a little part of `tcp_recvmsg`, which merely does the eventual copy of data, but apparently it represents the major time of handling an incoming tcp packet: figure 5 shows the first level of functions called by `tcp_recvmsg`: the whole time for these functions is shown (including their own calls to other functions), the remaining time is hence `tcp_recvmsg`'s own computation time.

1.3 Issue

Probes used to record their data in a big static kernel buffer, which would be flushed by a user daemon after the measurements were completed. However, this limits the recording

to the amount of physical memory, thus limiting its duration to only a few seconds on busy systems. We wanted to see data flushed while it is being produced.

2 General scheme

We had contradictory constraints: we would like to utilize a user program to manage the output, but we still want efficiency. Here are some solutions:

- The user program may use a special system call to get a block of data, and then use `write()` to flush it wherever it wants; this is the solution used by the Linux Trace Toolkit (see [2]), but the cost of switching between kernel and user level just doing the actual `write()`s may be too high on heavily loaded systems.

- The user program may open the output file, `mmap()` it and give the address to the kernel through a special system call, so that probes may directly write their data in the page cache, the flushing hence being automatic. However, `mmap()`ed files can't be extended, so that this would indeed increase the record limit up to the addressing limit, but it still fixes it.

- The user program may give the file name to a special system call, or as a parameter on module load, which a kernel thread would open and flush data to. This lets the kernel thread choose the most efficient way to achieve it and yet lets a user program control the output.

Starting from version 2.2, the Linux kernel supports the `sendfile()` system call for achieving good performance on an ftp server sending a file on the network for instance: the application gives the input and output file handles, the offset in the input stream and the size, `sendfile()` manages to do the transfer itself, as efficiently as possible. This avoids any `read()`-then-`write()` loop, along with all user/kernel level copies and switches. Actually, `sendfile()` might be renamed `copyfile()` someday, see section 5.

The solution we adopted is hence to load a mere block device driver in the kernel which defines its own sendfile method[2]. The recording user program, `fkt_record`, will open the block device (`/dev/fkt`) as a fake input stream, call some `ioctl()`s on it to setup the recording (buffer size, active mask,...), open whatever output file it wants, and finally call `sendfile()`. The sendfile method may then flush the probes' data to the output file as efficiently as possible. The size given to sendfile is then only limited by the amount of space on the destination drive. The recording lasts until the size limit is reached or a signal is received (for instance: `SIGCHLD` on termination of the traced test program, or `SIGINT` by pressing Ctrl-C).

[2]kernels prior to 2.5.30 need a 3-line patch in `common_sendfile()` to have it work

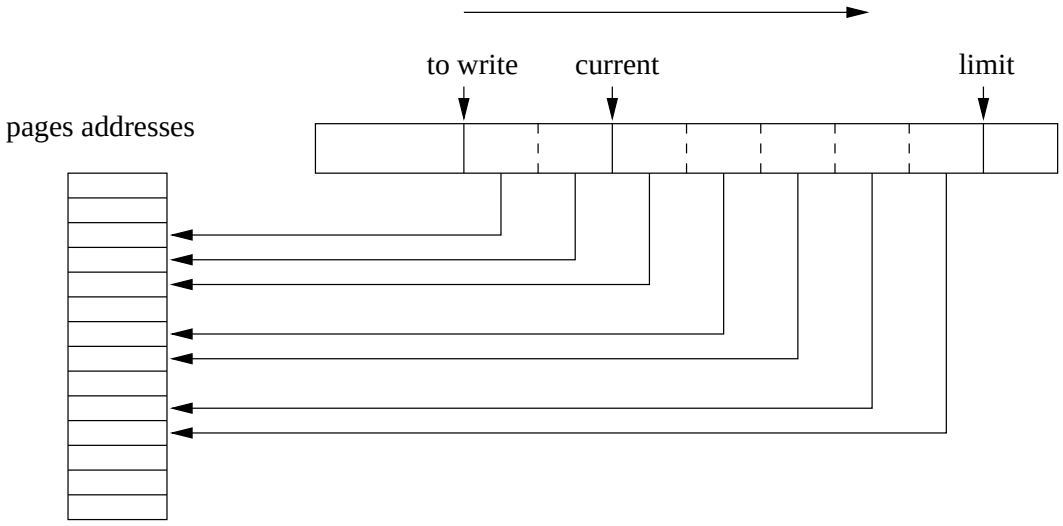

Figure 6: Rotating buffer

3 Buffer handling

We now need to modify FKT's buffering mechanism: since we want to be able to feed the page cache directly (see section 5), we shouldn't merely use a big buffer as FKT used to. Instead, we allocate pages in a bunch which may grow if needed, on load spikes for instance. Then we define a rotating buffer of pointers to the pages that probes may use to record data: see figure 6.

Pages are queued at the head (`limit`), probes write their recording data in the `current` page and switch it to the next one when filled up. Pages are then dequeued for writing (`to write`) until `to write` reaches `current`. Pages should generally be queued in the physical addressing order, because this lets `ll_rw_blk()` gather write orders and hence improve flushing. But if we lack pages, we might still want to break this order, at least temporarily, the goal being to avoid having to allocate pages on the fly which would perturbing the measurement. We then always try to re-enqueue pages (when the actual flush is achieved) in the physical order.

4 Synchronization

We now have two synchronization issues on SMP (Symmetric Multi Processing) machines: not only shouldn't probes use the same record slot (which was already addressed in FKT), but they should now properly switch pages when needed. The recording kernel thread

1. Read both pointers
2. Lock the page (atomically incrementing its counter)
3. Take some room: increment the pointer within the page, and if it goes out from it, increment the page pointer and take the room at the very beginning of the new page.
4. Try to cmpxchg both pointers:
   - if it fails (another CPU has modified the pointers since step 1), unlock the page and restart at step 2 (the updated value is given by cmpxchg)
   - if it succeeds,
     - if we incremented the page pointer, lock the new page and then unlock the old one
5. Fill the slot
6. Unlock the page (atomically decrement its counter)

Figure 7: Synchronization algorithm

must also avoid writing a page too early after a switch without waiting for probes which got a slot in the previous page but hadn't the time to fill it yet.

Synchronization between probes was already solved in FKT thanks to the 386 special instruction `cmpxchg` (see [7]) : it reads a memory location, compares it with a register, and if both are equal, writes a new value. With the `lock` prefix, this operation is done atomically, so that we are really sure to replace the old value by the new one, since no other CPU can write during the instruction. A probe would hence loop reading the current pointer, adding some room for its own use, and trying to do the `cmpxchg`, until the latter succeeds (i.e. no other CPU modified the pointer between the read and the `cmpxchg`).

The Pentium instruction set defines a `cmpxchg8b` instruction which does the same thing as `cmpxchg`, but 8 bytes at a time. This is sufficient for atomically updating two pointers at a time, provided they are adjacent in memory: the `current` pointer in the rotating buffer, and another one within the corresponding page.

We also have a synchronization issue between probes and the recording thread, because once they get their slot, probes take their time to write their recording (although it's generally quite short). The adopted solution is to have one counter per page, that every probe using this page would atomically increment, and only decrement once the recording is done. Therefore, the recording thread merely has to check that the counter is 0.

The final synchronization algorithm of a given probe is shown in figure 7. It was completely implemented in assembly language to maximize its speed and to be able to use it from anywere in the kernel.

It should be noted that on page switch, only one probe eventually does increment the page pointer, the one which gets the slot at the beginning of the next page.

Figure 8 shows the states that a probe goes through to get its slot. Its actions are in bold letters and the events coming from other CPUs are in italic letters. Only two logically consecutive pages of data (▭) are represented. The probe can lock (**P**) any of them, which is then noted with an L, or free (**V**) it. The current pointer in the pages is noted as ↓. The value the probe has read at the beginning or whenever a CmpXchg failed, and which it will use for the CmpXchg operation is noted as ↓. If they don't match (when another CPU achieved some CmpXchg in the meanwhile, which makes the current state move to the right side of the figure), CmpXchg fails and the algorithm has to restart (almost) at the beginning. The slot that the probe is trying to get is between two ↓, the actual computing of these pointers being noted as **Room**, or **RoomS** when it involves switching to a new page. Once a slot has really been obtained, it is noted as ▨.

Any probe which achieves a page switch also does two things. It writes at the beginning of the previous page the size of data recorded in it so that the analysis tool will know the padding length. It also wakes the recording thread up. It doesn't do this directly since this is some cost (it might even spinlock), but sets a softirq instead, i.e a mere bit in memory. On the return of any IRQ, the softirq function will actually wake the recording thread up, which will handle the freshly filled page.

5 Page Cache feeding

We did not describe how pages are actually written to disk yet. The `generic_sendfile()` function assumes that the destination file has a `sendpage()` method which just writes the page it is given. `generic_sendfile()` hence only needs to call `do_generic_file_read()` with an actor[3] which calls the `sendpage()` function.

However, `sendpage()` is not supported for real file systems yet. Kernels before 2.5.26 would simulate it through a mere `write()`, just like if the application called `write()`. Data would then be uselessly copied from a page to another within the page cache, but user/kernel switches are avoided. Starting from 2.5.26, the `sendfile()` system call would instead always return `EINVAL` because of a deadlock with `kmap`, corrected since.

Linus Torvalds explains in [3] (January 2001) that a real general `sendfile()` system call with no extra intra-cache copy should be possible with some work. A more recent post ([4], November 2002) even tells about renaming it into `copyfile()`, but we couldn't find any more recent news, apart from the implementation of the `sendpage()` method in many networking file systems.

As a consequence, we have to do the optimization ourselves. When a data page is ready, it is inserted in the page cache. We then call the simulation of `sendpage()`[4]. Since `write()` tries to fetch the destination page from the page cache, it gets the very page we

[3] the function which is called once the reading is successful

[4] which can now be added on recent 2.5 or 2.6 kernels since the `kmap` bug was corrected

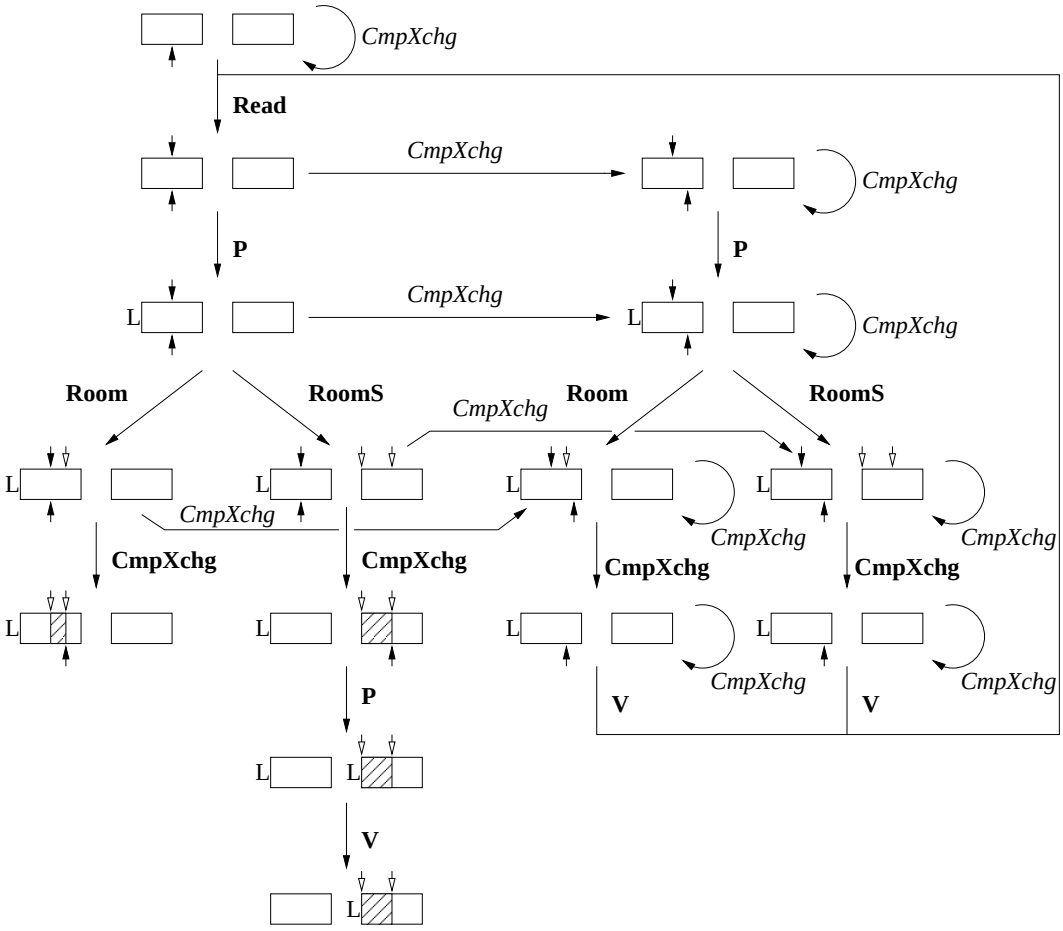

Figure 8: Synchronization

inserted, and a mere test on the address lets it discover that no copy is needed, all it needs to do is the usual size checks for instance and queuing the page to be written. We then periodically have a look at the page to see whether it was written. If so, it is removed from the page cache and can be queued back in the rotating buffer.

The real issue is checking pages in and out of the page cache. We added functions to `mm/filemap.c` and `fs/buffer.c` to do this (see section B.2 and B.3)

5.1 Checking in

Checking in is done by our new function `checkin_cache_page()`, which is much like `add_to_page_cache_unique()`, i.e. first checking that the page doesn't exist yet, setting some flags, and adding it to both the inode queue, the hash queue and the least recently used queue. But we also mark the page as Reserved so that it won't be swapped out, as Uptodate since the data is already in memory, and as Dirty to add it to the dirty pages queue, since it wasn't written on the disk yet. If the page was already in the page cache (which may happen if some process is reading the output file, although it is really not a good idea), checking in fails, and no optimization will happen.

Then we call the `sendpage()` method (actually `generic_file_write()`):

- On non-journalizing file systems, ext2 for instance, or on a plain disk partition, it will create buffer headers pointing inside the page, marked uptodate. Then it marks them as dirty and enqueues them on the dirty buffers queue. When bdflush is woken up, the buffers get scheduled for writing on the disk, i.e. they are locked, a flag is set to remember that the request was issued, and the order is eventually issued to the disk controller. When the hard disk reports that the data was actually written on it, buffers are unlocked.

- On journalizing file systems such as ext3 in write back mode, the page gets journaled. When kjournald is woken up, data are flushed the same way.

But these are not woken up that often, and we sometimes need to force a flush (see section 8) by calling the `fsync()` method of the file, which will schedule the data writing, according to the file system rules.

5.2 Checking out

Once a page is written, it can be checked out from the page cache. At the end of the recording, we also need to check out all the remaining pages before leaving the recording thread.

To know whether the page's buffers have been written, the recording thread has to check for each one that

- the write request has really been issued to the disk,

- it is not locked,

in that order, since the first statement assures that someone took care of the buffer, and since it had to lock it before setting that the write request was issued, we merely need to check that the buffer is not locked anymore, i.e. the write completed.

Then it can call `checkout_cache_page()` which locks the page and frees its buffers. Then it can safely remove it from the inode queue and the hash queue, and finally unlock it and clear the Reserved flag.

6 Analysis tool

Allowing long recording actually uncovered many rare odd situations that the analysis tool didn't handle properly because it would hardly ever happen with short recordings. For instance:

- The analyzer used to handle the special probe SWITCH_TO, which lives in the `switch_to()` function just before switching between processes, as an indication that the process has changed. But of course an IRQ may (rarely, though) be raised just between the probe run and the actual switch, so that the current pid recorded by probes of the IRQ handler is the same as before, upsetting the analyzer. Now it considers that the switch really happened when a probe is run for the new process.
- On switching between process, it also used to take the time of the SWITCH_TO probe as restarting time for new process. Not only is this wrong since the former process may have handled an IRQ in the meanwhile, as described above, but it may even have already been scheduled on another CPU, in which case the last SWITCH_TO probe doesn't have any meaning any more (and probably don't exist any more). It now uses the time of the last probe on that CPU.
- It also used to consider calls to a particular function as a stack, but this is wrong for functions that sleep on UP (Uniprocessor) machines, and for any kernel function not protected by locks on SMP machines. It now uses per-process stack.
- The idle process is a special case, since there is actually one per CPU, with the same special pid 0. It works as long as only one CPU is idle at a time, but else the statistics are wrong. It now transforms the special pid 0 into the CPU number, negated, so that even per-CPU idleness can now be seen.

Some new features are also desirable:

- When tracking a bug on long runs, only the small portion when the bug happened really needs analyzing. We developed a tool which lets the user view the activity of the system during the trace, in terms of how fast probes filled pages. Figure 9 shows a 3 minute run with several attempts to trigger a bug, which eventually happened in the middle of the trace. The area of the bug (a dozen seconds) could then be selected, and extracted with another tool, for a much faster analysis than analyzing the whole trace.

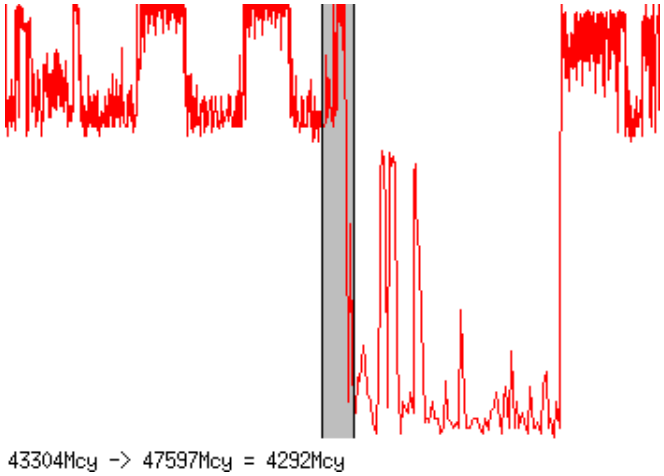

Figure 9: selecting a part of a trace

- On runs with several different processes, it might be hard to remember the pid of each process. Now, before recording, `/proc/*/stat` are read to get the command name of each process and written before the trace. Then, a `do_fork` probe was added between choosing a new pid and scheduling the new process, and a `do_exec` probe was added with the new command name as parameter, so that the analysis tool will be able to follow processes names and print them.
- On long runs with a lot of processes, a kernel compile for instance, the pid number may wrap, and the analyzer would mix the statistics of processes with the same pid. It now uses a special probe in the `unhash_pid()` function, which is called after checking that the process is not running and not to be ever run again, but before releasing the spinlock which lets `fork()` choose a new pid, hence avoiding any collision.

7 Automatic tracing

Adding probes by hand allows one to ask for parameters of the function call, computed values, and the return code at will, but it might be a drudgery to add them to every function of a `.c` file. The `-finstrument-functions` gcc option will actually do the work: it will add a call to `__cyg_profile_func_enter()` on entry in and `__cyg_profile_func_exit` on exit from any function. It can hence be used to quickly add tracing to a set of `.c` files.

The drawback is that it only gives the address of functions and the address of caller sites. The number of parameters is unknown, and there is no way to get the returned value. It should hence only be seen as a beginning, and if some parameters or returned values are

needed for a particular function, the `no_instrument_function` attribute can be set for it, and then probes be added by hand.

The other drawback is that no probe code is given, so that the analysis tool only gets function addresses. The file `System.map`, built at compilation time, gives the address of every function built in the kernel. There is no way to get it automatically, so the user needs to give it. But this is not sufficient for modules, which don't appear there since the actual address in memory is not known at compilation time. `/proc/kcore` holds the addresses of exported functions, i.e. those which are available to modules, as well as the loaded modules addresses and addresses of their exported functions, for the current running kernel. This is safer than relying on the user to give the proper `System.map`. However, it doesn't hold addresses of non-exported functions which could still call `__cyg_profile` functions.

`/proc/kcore` and `System.map` are hence merged and included in the record file by the user-land recording program `fkt_record` before launching the measurement. Any incoherency will show whether the user provided the proper `System.map` file.

Addresses of non-exported functions from modules are still missing, though. `/proc/kcore` gives the base address of every loaded module as well as the path to its `.o` file, so `fkt_record` runs `nm` on these files to get functions addresses offsets and add them to the module bases, hence getting every function address.

8 Performances discussion

8.1 Managing actual writing on disk

The goal is to perturb the System behavior as little as possible, and still have data flushed as soon as possible to have free pages for probes.

By default, the kernel won't launch write orders as soon as data is submitted, which is good, since it permits gathering orders for adjacent blocks and issuing them at the same time, executing them on the same spin of the disk for a given cylinder. Actually, it won't do it at all until memory begins to be filled up, or until some time has passed. The `bdflush` kernel thread (or `kjournald` for journalized file systems) has the responsibility to enforce this by launching write orders whenever it is woken up. Since it is woken up on a periodical basis and by memory allocation functions when memory becomes tight, the goal for every day use is achieved.

Our problem is that we don't want to fill memory at all, but use the pages we allocated once for all. `bdflush` might then be called too late, and all of our pages get filled up before any write order is launched.

We don't want to wake up `bdflush` ourselves either, because it would launch a bunch of writes from the whole buffer cache, which raises two issues:

- it changes the behavior of the system concerning issuing write orders from other processes, which might cause them to behave differently (and we may want to trace their normal behavior);

- it may actually not launch our pending writes immediately, if another process asks for a lot of writes, for instance.

Instead, we can ask the kernel to synchronize the dirty buffers of our trace file thanks to the `fsync()` method. But even in 2.6 kernels, there is no non-blocking method which would permit an asynchronous synchronization yet: `fsync()` will not return until the list of buffers which were dirty just before the call are really written on the disk.

To handle with this, we launch a synchronizing thread at module insertion, which will call `fsync()` whenever the recording thread asks it to. Only this thread will have to wait for the list of dirty buffers to be really written, while the recording thread will be able to reuse pages as soon as they are really written to disk without having to wait for them as a whole. Figure 10 shows this:

- pages filled by probes are shown in black,

- pages that have been submitted by the record process but whose write orders were not launched yet are shown in blue, and follows the black line quite well (whenever the recording thread is scheduled),

- really written pages are shown in green,

- and by adding the number of allocated pages (in yellow, always 128 pages in this example), we get

- the available pages in red, which the black line shouldn't collide with.

To prevent the black line from colliding with the red one, the recording thread will ask the synchronizing thread to launch a synchronization whenever the number of available pages falls down to three-quarters of the allocated pages. Writing on disk might be too slow however, so if the ratio falls down to one-eighth, extra pages are allocated until it raises up to one-quarter. If this is really not sufficient and a probe doesn't find space for its recording, it will set the active mask to zero, hence pausing the measurement, and a warning is printed. These threshold ratios are quite arbitrary, and a tuning might be needed.

8.2 On the user side

The user also needs to be aware of flushing issues.

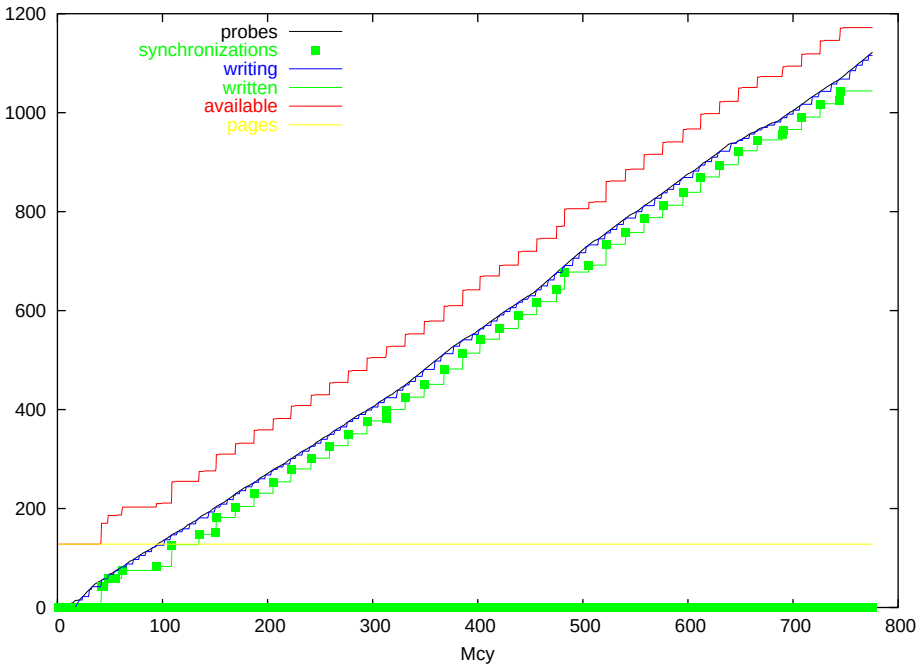

Figure 10: data flushing

8.2.1 Extreme situations

For instance, adding a probe to the function that sends a byte on a gigabit network is prone to have the recording stopped very early, unless the disk where the trace is recorded can achieve gigabytes per second transfers!

Probes which may be encountered when writing the trace pages are also of much concern: if, for writing a one trace page, encountered probes fill more than another trace page, there will be just more and more dummy trace data!

Of course, these are extreme situations. Usually, the issue is just avoiding perturbing measurements too much.

8.2.2 Advice

The load of the recording thread during the measurement can be found just like any other process load: in the analysis summary. It can be as small as a few tenths of a percent, but may raise up to 5% for instance. By properly choosing the precise layers of probes that need to be recorded, thanks to the active mask, the load of the recording can be tuned.

When stressing a part of the kernel, the path of the recording shouldn't be stressed, since it would prevent good flushing. When stressing a SCSI driver, an IDE disk should be used for recording for instance.

The recording may be done on any file system, but a file system well suited to bulk writing should be preferred. The best thing to do is even avoiding any file system and writing onto a plain partition.

9 Conclusion and future work

This work has enhanced an existing tool for tracing the Linux kernel by completely redesigning its recording mechanism.

We addressed a time-critical synchronization issue on SMP systems and wrote the synchronization algorithm in assembly language to minimize the probe cost.

Modifications were made to the page cache to avoid any copying of the recording buffers.

We also had to dig quite deeply into the kernel flushing mecanism to understand how it works exactly to avoid perturbing it while recording. This has shown how the Linux Cross-Reference site [8] and the usage of C-generated tags in an editor can be useful for browsing in the kernel source.

New issues were raised by the mere fact of now having very long measurements: apart from some bug fixes, new tools were designed to display the recording activity and allow interactive selection of a part of a long recording session for precise analysis.

Work was also done to be able to trace a module without any need to manually insert probes in the module source file.

This improved version of FKT has been used to trace another project of Pr RUSSELL, iSCSI. It has proven useful to understand some memory issues, for instance, and to improve performance. It also generated the ideas for the new tools.

What is left to do is to develop a complete graphical interface that would integrate all of these tools.

A Technical information

A.1 Hardware

A.1.1 Achernar, Capella, Hadar, Procyon

CPU/RAM: PIII 600Mhz 256MB

Ethernet: 100BaseTX: 3Com PCI 3c905B, 100BaseTX: Digital DS21140 Tulip, Gigabit: 3Com 3c985

OS: RedHat 9, kernel 2.4.21

A.1.2 Nudra, Ophiuchi

CPU/RAM: Bi PPro 200MHz 128MB

Ethernet: 100BaseTX: 3Com PCI 3c905B, 100BaseTX: Digital DS21140 Tulip

OS: RedHat 9, kernel 2.4.21

B Additions to the Linux Kernel

Some modifications needed to be done to integrate FKT into the kernel. Then the recording code could be loaded as a module, which facilitated development a lot.

B.1 System calls

Added 3 system calls: `sys_fkt_probe0()`, `sys_fkt_probe1()` and `sys_fkt_probe2()`, to allow user level applications to be traced as well.

B.2 Buffer cache

Added to `fs/buffer.c`:

`buffers_dirty()` which tells whether buffers of a page have been written to disk.

`wait_buffers_clean()` which waits for a page's buffers to be written to disk.

B.3 Page cache

Added to `mm/filemap.c`:

`checkin_cache_page()` which brings a page into the page cache for a given file mapping, at a given index.

`checkout_cache_page()` which takes a page out from the page cache. If the page is dirty, it is cleaned. After that, the page can be reused for anything else.

B.4 FKT module

The new FKT module, `fkt-mod.o`, manages two buffers:

- the pages buffer (`pagebuf`), which holds addresses of the allocated pages. It allows to consider them as a bunch, even if they are not all contiguous in memory.
- the rotating buffer (`rotatingbuf`, see section 3), which holds pointers in the pages buffer as a FIFO stack. It also holds one lock per enqueued page.

Some macros were defined to easily manage pointers in these buffers: `PAGESUCC`, `PAGEPRED`, `PAGEINC`, `PAGEDEC`, and `ROTSUCC`, `ROTPRED`, `ROTINC`, `ROTDIFF`.

Some functions were defined to allocate and free pages: `fkt_alloc_pages()` tries to allocate a bunch of consecutive pages, with some extra checks. `alloc_buffer()` allocates a bunch of pages, first trying to get them as a consecutive buffer, and if this fails, calling itself recursively to get two bunches of half the size. This is intended to get as many consecutive pages as possible to maximize gathered writes on disk. `free_buffer()` frees them, with some extra checks to ensure that the page cache didn't pinned them for another usage.

The synchronizing thread was defined as a single function `fsync_thread()`, which is spawned at module insertion and aborted at its deletion. It keeps sleeping, and if woken up, looks for a `struct file *` in `fsync_file`, and if found, flushes it. See section 8.1.

`sendpage()` was then defined as an FKT temporary replacement for any future generic `sendpage()` method (see section 5). It calls `checkin_cache_page()` to check the page into the page cache and calls the actor (i.e. `generic_file_write()`).

The `sendfile()` method for FKT was finally defined, it will:

- acquire the FKT lock,
- allocate the pages and rotating buffers if not done for a previous recording,
- allocate data pages and record their addresses in the pages buffer if not done for a previous recording,
- initialize pointers, enqueue data pages in the rotating buffer,

- let probes record: set the active mask to the configured `initmask`,
- wake up waiting processes, see `ioctl(FKT_WAITREADY)` below,
- loop while the limit size is not reached and no signal is pending:
  - sleep, and when woken up:
  - issue write orders, i.e. dequeue data pages and call `sendpage()` for them
  - re-enqueue consecutive written data pages in the rotating buffer,
  - force a synchronization if the rotating buffer hasn't many pages left, by setting `fsync_file` and waking up the synchronizing thread,
  - if the rotating buffer really has few pages, enqueue written data pages, even if not consecutive, and if this is not sufficient, allocate some extra pages.
- stop probes,
- if the limit size was not reached, issue pending write orders,
- force a synchronization,
- wait for last writes to complete,
- return the recorded size.

We also added an `fkt_ioctl()` function which permits some configuration and synchronization from user-land by calling the `ioctl()` system call on `/dev/fkt`:

- `FKT_SETINITPOWPAGES` configures the number of initially allocated pages. This can be useful to have fkt allocate a lot of memory, if high load spikes are expected during the measurement.
- `FKT_SETTRYDMA` asks fkt to get pages in the DMA capable area if possible. This is only useful if the Disk drive is connected to an ISA card, for which DMA is only possible with addresses below 16M.
- `FKT_FREEBUFFER` asks fkt to free its allocated pages. Fkt only deallocates its pages when explicitly asked to, to avoid losing the consecutiveness.
- `FKT_WAITREADY` will hang until the `sendfile ()` function has done all of its initialization.
- `FKT_ENABLE`, `FKT_DISABLE`, `FKT_SETMASK` will set a bit of, clear a bit of, or even set the active mask, hence giving the possibility to trace different layers during the measurement.

B.5 Probes

Probes were already inserted in 8 different layers, differentiated by the key mask:

System: on entry and exit of every System call, trap or IRQ,

Network drivers: for 3c95x, tulip and acenic cards, on send and receive functions,

Network layer: IP route, send and receive functions,

Transport layer: UDP and TCP send and receive functions,

Socket layer: Inet and generic socket functions,

Scsi layer: generic command functions, disk and generic SCSI drivers,

File System layer: `read()`, `write()`, `lseek()`, `fsync()`, `nanosleep()`,

Application layer: `sys_fkt_probe0()`, `sys_fkt_probe1()` and `sys_fkt_probe2()` system calls from applications.

We added some new probes in 2 new layers:

Gcc instrumentation: automatic gcc instrumented functions, see section 7,

Fkt: fkt-related codes, for instrumenting itself and to get the figure 10.

C FKT Tools

C.1 Record file format

The record file got enriched with much new information about the machine on which the recording was done. Now it includes the following:

Miscellaneous: (as in previous version) start and end time of recording, number of CPUs, pid of the recording process, and of the traced process,

Irqs: (as in previous version) for each IRQ, the devices associated to it,

Uname: the equivalent of `uname -a`: the machine name, the version of the running kernel, its compilation number,

Symbols: all available information about function addresses, for gcc instrumentation, see section 7,

Pids: the command names of every running process before the recording,

and finally the recording itself, as consecutive pages of probe slots.

C.2 fkt_record

`fkt_record` is used to run a recording session. For instance, running `fkt_record ls` will trace the execution of the `ls` command and write the output record, as described in the previous section, in file `trace_file`. Some options were added to handle the new features, in addition to the `-f <file>` options which permits to choose another name than `trace_file`.

`-k mask` allows to choose the key mask which will be set for the recording. It defaults to to `0x1`, i.e. only system calls, IRQs and traps are traced, but setting other bits allows to trace other layers, see section B.5,

`-s size` restricts the maximum recording size. By default, the available size of file system or the block device will be used,

`-n` prevents `fkt_record` from spawning any process. One can then separately launch a script which may set bits in the active mask, launch some commands to be traced, set other bits, etc... And have all the trace in one file. The trace ends when Ctrl-C is pressed. To set and clear bits of or set the active mask, `fkt_enable`, `fkt_disable` and `fkt_setmask` programs were written, which merely call the corresponding `ioctl()` on `/dev/fkt`,

`-S System.map` will merge the given file with `/proc/kcore`, see section 7.

`-p pow` will have fkt initially allocate 2^{pow} data pages for the recording. The default is 7, ie 512KB.

C.3 fkt_print

Once the recording is done, it can be analyzed by `fkt_print`, even on another machine, since the record file includes sufficient information (see section C.1). For instance, `fkt_print`, called as such, will dump the recording from `trace_file` while analyzing it, and finally display some statistics. The formatting of the output was much revisited to be able to display large numbers, but its content is still the same, apart from the corrections of section 6. Some options were added:

`-s pid` asks it to only compute statistics when the given process is alive,

`-p pid` asks it to only print the record dump when the given process is alive, or even only print the record dump for this process if '+' is prepended to the pid,

`-k` will exclude user-level computation time from the final histogram, as shown on figure 4,

`-b trace.bufdat` will dump some data about FKT's flushing in the given file. The corresponding layer (see B.5) must have been included in the active mask. By running `fkt_bufstats` on it, one can get figure 10,

`-t trace.timedat` will dump the record pages filling times in the given file. `fkt_selection` can then be used to display it (see figure 9) and select a part of the recording, `fkt_extract` can then be used to extract it for much faster analysis.

References

[1] Robert D. RUSSELL , Mrinalini CHAVAN, *Fast Kernel Tracing: A Performance Evaluation Tool for Linux*. Proceedings of the 19th IASTED International Conference on Applied Informatics, Innsbruck, Austria, 19-22 February, 2001.

[2] Karim YAGHMOUR, *The Linux Trace Toolkit*, `http://www.opersys.com/LTT/`

[3] Linus TORVALDS, in thread *Is sendfile all that sexy?*, Linux Kernel Mailing List, `http://www.cs.helsinki.fi/linux/linux-kernel/2001-02/0958.html`

[4] Linus TORVALS, in thread *NFS mountned directory and apache2 (2.5.47)*, Linux Kernel Mailing List, `http://groups.google.fr/groups?selm=ar3nrr%24f1a%241%40penguin.transmeta.com`

[5] Alessandro RUBINI & Jonathan CORBET, *Linux Device Drivers, 2nd edition*, O'Reilly (2002)

[6] Daniel P. BOVET & Marco CESATI, *Understanding the Linux Kernel, 2nd edition*, O'Reilly (2003)

[7] *Intel Architecture Software Developer's Manual*, Volume 2: Instruction Set Reference

[8] *The Linux Cross Reference*, `http://lxr.linux.no/`

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803