The New Linux 'perf' Tools

Arnaldo Carvalho de Melo
Red Hat Inc.
Brazil
acme@redhat.com, acme@ghostprotocols.net

1 Abstract

The perf events infrastructure is fast moving into being the unifying channel for hardware and software performance analysis.

Modern CPUs have hardware dedicated to counting events associated with performance, special registers that allow pinpointing hotspots that can possibly be optimized.

Recent developments in the Linux kernel explore these features, solving problems found in previous attempts, such as OProfile.

The reuse of Linux kernel code in user space applications as part of the experiment of shipping user space tools in the kernel repository is discussed.

The record, report, annotate, diff, probe, test, and tracing tools, among others that are in the works, are described.

Throughout this paper the current status of development will be discussed, mentioning ongoing discussions and feedback to decisions taken, trying to help interested people in joining the effort to have better hardware and software event processing tools in the Linux environment.

2 Development Model

To avoid the disconnect of the kernel and user parts for a tool directly used by the kernel community the perf tools are being developed in the kernel repository, a first.

One example of this disconnect can be read in a bug tracking entry[1] where Linus Torvalds reported not being able to use the hardware performance counters on recent hardware and kernel. The problem was that the userspace code to match the kernel support was not present on the OS used.

While this happens sometimes with other OS components the nature of profiling being so close to kernel development lead to this attempt to develop both together.

That does not mean that there is not an ABI guarantee, if it changes in a non backward compatible way, it is a bug. There are already several tools that are not hosted in the kernel repository that depend on the perf ABI, sysprof for instance[2].

Another goal is to reduce the barrier for coding in the kernel and in the user space tools, so the tools are being developed following kernel coding style and conventions, reusing the familiar list, rbtree, error reporting, hash facilities and others as the experiment matures.

3 Tools Usage Model

In the model used there are no daemons, instead there is a growing number of sub-commands implemented in a single binary, very much like the git tool, from which code was borrowed, keeping features like the automatic pager.

The tools can be used by non root users, with operations being allowed only for threads they have permission to monitor.

In the next chapters such commands will be described, in a order that is the most common in simple profiling sessions, then other commands for creating dynamic probes, listing the available events, querying and manipulating the debug cache and others will be presented.

4 List

The list of events that can be specified can be found using the 'list' tool. By default the tools uses the `cycles` hardware counter, if it is not available, the `cpu_clock` software counter is used as a fallback.

cpu-cycles OR cycles
instructions
cache-references
cache-misses
branch-instructions OR branches
branch-misses
bus-cycles

Table 1: Hardware events.

cpu-clock
task-clock
page-faults OR faults
minor-faults
major-faults
context-switches OR cs
cpu-migrations OR migrations
alignment-faults
emulation-faults

Table 2: Software events.

rNNN (see 'perf list –help' on how to encode it)	Raw hardware event descriptor
mem:<addr>[:access]	Hardware breakpoint

L1-dcache-loads
L1-dcache-load-misses
L1-dcache-stores
L1-dcache-store-misses
L1-dcache-prefetches
L1-dcache-prefetch-misses
L1-icache-loads
L1-icache-load-misses
L1-icache-prefetches
L1-icache-prefetch-misses
LLC[1]-loads
LLC-load-misses
LLC-stores
LLC-store-misses
LLC-prefetches
LLC-prefetch-misses
dTLB-loads
dTLB-load-misses
dTLB-stores
dTLB-store-misses
dTLB-prefetches
dTLB-prefetch-misses
iTLB-loads
iTLB-load-misses
branch-loads
branch-load-misses

Table 3: Hardware cache events.

5 Top

The first tool is a `top` like that provides snapshots of system activity according to a specified event, or `cycles` if none is specified.

It will decay the percentages over time, allowing to see transitioning shift on symbols sampling.

Dynamic symbol annotation is available by pressing the `s` key, where the sorted by sample list of symbols display is replaced with the function disassembly with per line sample percentages.

The default shows symbols for kernel and user space samples, but one or the other can be selected via command line or special keystrokes.

This tool is usually the first one used to get a quick glimpse of what is happening on the system.

```
-------------------------------------------------------------------------------
PerfTop: 406 irqs/sec kernel:42.6% exact: 0.0% [1000Hz cycles], (all, 2 CPUs)
-------------------------------------------------------------------------------

samples  pcnt function                                    DSO
_______ _____ __________________________________________ ___________________

 268.00 13.6% pixman_rasterize_edges                      libpixman-1.so.0.14.0
 255.00 12.9% read_hpet                                   [kernel]
  75.00  3.8% do_raw_spin_lock                            [kernel]
  50.00  2.5% SelectorMatches(RuleProcessorData&, nsCSS libxul.so
  49.00  2.5% js_Interpret                                libmozjs.so
  43.00  2.2% strstr                                      /lib64/libc-2.10.2.so
  39.00  2.0% __pthread_mutex_lock_internal               libpthread-2.10.2.so
  32.00  1.6% fget_light                                  [kernel]
  32.00  1.6% __pthread_mutex_unlock                      libpthread-2.10.2.so
  27.00  1.4% dso__load_sym.clone.0                       /home/acme/bin/perf
  25.00  1.3% fput                                        [kernel]
  24.00  1.2% acpi_os_read_port                           [kernel]
  24.00  1.2% IA__g_main_context_check                    libglib-2.0.so.0.2000.5
  24.00  1.2% native_read_tsc                             [kernel]
  22.00  1.1% vte_terminal_match_check_internal           libvte.so.9.6.0
  22.00  1.1% _raw_spin_lock_irqsave                      [kernel]
  19.00  1.0% _raw_spin_unlock_irqrestore                 [kernel]
  19.00  1.0% schedule                                    [kernel]
  18.00  0.9% do_select                                   [kernel]
  16.00  0.8% unix_poll                                   [kernel]
  15.00  0.8% ioread32                                    [kernel]
  15.00  0.8% hpet_next_event                             [kernel]
  14.00  0.7% do_raw_spin_unlock                          [kernel]
  14.00  0.7% nsPropertyTable::GetPropertyInternal(nsPr libxul.so
  12.00  0.6% __memchr                                    /lib64/libc-2.10.2.so
  12.00  0.6% symbol_filter                               /home/acme/bin/perf
  10.00  0.5% do_sys_poll                                 [kernel]
  10.00  0.5% copy_user_generic_string                    [kernel]
  10.00  0.5% imgContainer::DrawFrameTo(gfxIImageFrame* libxul.so
   9.00  0.5% free                                        firefox
   9.00  0.5% SearchTable                                 libxul.so
   8.00  0.4% nsRuleNode::GetStyleDisplay(nsStyleContex libxul.so
   7.00  0.4% SelectorMatchesTree(RuleProcessorData&, n libxul.so
   7.00  0.4% __pollwait                                  [kernel]
   7.00  0.4% ehci_work                                   [kernel]
```

6 Stat

To just count the number of events that took place while some workload runs the `stat` tool can be used.

Multiple events can be specified, as in the default case:

```
$ perf stat sleep 1

 Performance counter stats for 'sleep 1':

          1.041007  task-clock-msecs         #      0.001 CPUs
                 1  context-switches         #      0.001 M/sec
                 1  CPU-migrations           #      0.001 M/sec
               177  page-faults              #      0.170 M/sec
           1452619  cycles                   #   1395.398 M/sec
            703504  instructions             #      0.484 IPC
            148868  branches                 #    143.004 M/sec

       1.001832642  seconds time elapsed

$
```

The -repeat option can be used to print the average and standard deviation found after running the specified workload multiple times.

7 Record

Record specified events for the whole system, a CPU or some specific workload (threads started from the record session) for later, possibly offline, analysis.

Events can be specified on the command line, even in groups, where the tools requests that a set of counters must be scheduled at the same time or not at all.

The recording session can be for an existing process or thread, when the pid/tid must be specified so that samples can start to be taken for the desired entities.

It is also possible to ask for sampling the whole system or a specific cpu.

On architectures without hardware perf counters a fallback is made to the equivalent software based counter, using high resolution timers, still providing profiling according to the intent stated.

Callchain support can be specified when the kernel will traverse the frame pointer at each sample, stashing the callchain in the perf.data file together with the sampled event.

Most userspace applications and libraries are built without frame pointers, so as to use the extra register for optimizations. In register starved hardware architectures the performance impact prohibits using this scheme to collect callchains.

Investigation is underway to find the most efficient way of obtaining callchains for userspace when frame pointers are not available. DWARF unwinders in the kernel, collecting a few kilobytes of userspace stack if it is already present in memory, and using the BTS[2] in processors where it is present are schemes being considered.

At the end of the session, when the workload finishes or the tool is exited by pressing control+C, the events are scanned to find the list of DSOs in executable mappings where samples were taken. This list is then traversed to extract the build-ids for such DSOs. The resulting table is written to the perf.data file.

[2]Branch Trace Store

This is to allow later symbol resolution in the other tools by looking up the symtabs by the build-id, which allows precise matching of samples to symbols. In systems without build-ids a best effort, by name, lookup is done. This may be a problem for long running record sessions where it is possible that some DSOs get replaced in a system update, causing later matching to fail.

The record tool also maintains a build-id cache, storing the DSOs found on a directory hierarchy that is a superset of that used by debuginfo packages in Linux based OSes such as Fedora. It is a superset because it keeps a directory per DSO to allow different versions to be stored, keyed by build-id.

This allows use-cases that will be described in the diff section, where two perf.data files collected for different versions of a DSO can be conveniently compared.

8 Report

Post processor that reads files created with `perf record` and show the results sorted as specified by the user (comm, DSO name, symbol name, etc), using simple stdio based output or via a TUI that is integrated with annotation and callchain tree browsing.

The report tool uses the build-id table in the perf.data file to locate the right symbol table in the debug cache. The use of build-ids allow many use cases such as samples in one machine, using the record tool, without populating the debug cache and then transferring the perf.data file to a different machine, possibly of a different hardware architecture, with a debug cache containing the DSOs keyed by build-ids found in the perf.data file.

It is possible to specify the set of columns to sort the data, so as to have per thread views, per DSO views. Filtering is also possible, allowing one to zoom into a specific component of the workload.

All these operations can be done via the command line or using a text user interface based in the newt and slang libraries.

The TUI interface allows for quickly compounding zoom operations, per thread and per DSO. It is being designed trying to allow the common, fast path usage operations done in a profiling session, like going from the list of most sampled symbols to its annotation, as described in the annotate tool section later in this paper.

Another feature in the TUI that saves time when reading profiling data is that one does not have to first run `perf report`, locate some specific function of interest, exit and then run `perf annotate symbol-name`. The time to process the data file is shared by the `report` and `annotate` steps.

Example of a profiling session when writing this paper:

```
$ perf record make perf.pdf
===================== first latex run: perf.tex ===========================
===================== additional latex run: perf.tex ==========================
===================== first pdflatex run: perf.tex ===========================
===================== additional pdflatex run: perf.tex =========================
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.043 MB perf.data (~1890 samples) ]
$
```

```
$ perf report | head -25
# Events: 852  cycles
#
# Overhead    Command         Shared Object                            Symbol
# ........  .........  ....................  ............................
#
     4.93%      latex  libc-2.10.2.so        [.] fgetc
     4.47%   pdflatex  libc-2.10.2.so        [.] fgetc
     3.71%      latex  pdftex                [.] sarestore
     3.64%   pdflatex  pdftex                [.] sarestore
     2.39%   pdflatex  pdftex                [.] zshoweqtb
     2.25%      latex  pdftex                [.] zprintcmdchr
     2.16%      latex  pdftex                [.] zshoweqtb
     1.94%      latex  pdftex                [.] pseudoinput
     1.94%      latex  pdftex                [.] zshowsa
     1.91%      latex  pdftex                [.] fm_scan_line
     1.84%   pdflatex  pdftex                [.] zprintcmdchr
     1.36%      latex  libkpathsea.so.4.0.0  [.] hash_insert_normalized
     1.36%   pdflatex  pdftex                [.] zshowsa
     1.33%   pdflatex  [kernel.kallsyms]     [k] clear_page_c
     1.27%      latex  pdftex                [.] showactivities
     1.25%   pdflatex  pdftex                [.] convtoks
     1.25%   pdflatex  pdftex                [.] pseudoinput
     1.14%   pdflatex  libc-2.10.2.so        [.] __GI_strcmp
     1.14%      latex  pdftex                [.] zautoexpandfont
     1.03%   pdflatex  pdftex                [.] zpdfprintreal
$
```

9 Annotate

Shows source code and/or binary disassembly prefixed by percentage of hits per instruction/source code line.

This tool leverages the objdump disassembly mode familiar to developers while augmenting it by adding a new column with the percentage of hits for that specific line.

There is also a TUI interface that allows for quickly cycling through the most sampled lines in a function, with the tool starting by centering the screen on the most sampled line in the function.

It is integrated with the build-id cache, so it is possible to annotate functions for binaries in OS packages already updated, or in previous versions built in the lifetime of a project.

10 Diff

Compares multiple perf.data files recorded to show difference that different versions of workloads being analysed or differences resulting from changes in app/kernel specific tuning knobs.

The most basic use case is:

```
$ perf record ./code
$ ls perf.data*
perf.data
$ vi code.c
$ make
$ perf record ./code
$ ls perf.data*
perf.data perf.data.old
$ perf diff
```

By default it compares the samples in perf.data.old against the ones in perf.data.

Further ideas include plotting graphics from more than two files, say one per kernel release for some specific workload, to compare the distribution of metrics over time.

Paul McKenney describes several techniques that should be supported and discusses how such a tool can be used to pinpoint problems in his "Differential Profiling[3]" paper.

11 Probe

Dynamic probing using processor breakpoints or code changing to insert jumps, working with the other aspects of the perf infrastructure (callchains, report tools).

It works by creating new probes that then can be used by the other tools, as in this example:

```
[root@emilia ~]# perf probe schedule
Add new event:
  probe:schedule       (on schedule)

You can now use it on all perf tools, such as:

perf record -e probe:schedule -aR sleep 1

[root@emilia ~]# perf probe --list
  probe:schedule       (on schedule@sched.c)
[root@emilia ~]# perf record -e probe:schedule sleep 1
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.006 MB perf.data (~267 samples) ]
[root@emilia ~]# perf report
# Events: 3  cycles
#
# Overhead  Command  Shared Object                Symbol
# ........  .......  .............  ....................
    33.33%    sleep  libc-2.12.so   [.] __nanosleep_nocancel
    33.33%    sleep  libc-2.12.so   [.] __GI__dl_addr
    33.33%    sleep  ld-2.12.so     [.] dl_main
#
# (For a higher level overview, try: perf report --sort comm,dso)
#
```

Once a dynamic probe is created it can be used by various tools till its use is no loger needed, when it can be deleted with:

```
[root@emilia ~]# perf probe --del schedule
Remove event: probe:schedule
[root@emilia ~]# perf probe --list
[root@emilia ~]#
```

The tool allows as well specifying some absolute lines in the source code, relative lines from function start, function return, listing the source code for an specific function and other facilities.

12 Test

Suite of regression testing infrastructure for the perf kernel and user space libraries infrastructure.

Right now contains just a set of tests for the symbol library, checking among other things that the parsing of `/proc/kallsyms` matches what is obtained from the `vmlinux` ELF symbol table.

More tests like creating a synthesized workload in a thread to generate some number of events countable via hardware or software events and then creating counters to measure it, should be written, but the infrastructure for that is in place in the `perf test` command.

13 Tracing

There are several attempts of bringing ftrace and perf together, the current trace tool is one such attempt, but since it was introduced development on a separate userspace tool for ftrace, trace-cmd, took place.

There are many synergies, so there is ongoing discussion on how to bring those together. Probably trace-cmd will be merged somewhere in the tools/ directory and subsequently work will be done on bringing the trace libraries closer.

In the meantime several tools that combine using the record tool with a tracepoint event and specialized processing of the resulting traces collected in a perf.data file.

The kmem tool is an example, it records events for these tracepoints:

- `kmem:kmalloc`
- `kmem:kmalloc_node`
- `kmem:kfree`
- `kmem:kmem_cache_alloc`
- `kmem:kmem_cache_alloc_node`
- `kmem:kmem_cache_free`

It then processes the events using library infrastructure that is common to all tools, and processes the raw tracepoint events, keeping track of allocations and frees, to then provide a summary.

14 Recent Developments

RAPL[3] is a new feature which provides mechanisms to enforce power consumption limits on some new processors, it provides a new MSR[4] which reports the total amount of energy consumed by the package.

Work is being done by Intel engineers that exports this as the "energy" hardware counter[4]:

```
$ perf stat -e energy test
Performance counter stats for 'test':

        202     Joules cost by package
7.926001238     seconds time elapsed
```

15 Conclusion

The tools are advancing at a growing rate, more developers are joining the effort, but it is the actual use by developers that will ultimately shape these tools.

The team working on these tools tries hard to listen to users, celebrating each use case that gets reported, it is really important that feedback is provided to guide the work on providing these tools.

16 Acknowledgments

The author would like to thank Frédéric Weisbecker, Ingo Molnar, Mike Galbraith, Paul Mackerras, Peter Zijlstra, Stephane Eranian, Tom Zanussi and many other contributors, as well as all the people using and providing suggestions and criticism.

Much more can be written and documented about these tools about the live mode, the symbols library, the kvm and archive tools, several trace tools besides kmem, more than I was able to write in this paper, consider contributing on this front!

References

[1] Linus Torvalds, *Bug 497230 - oprofile doesn't like atom or Nehalem*, http://bugzilla.redhat.com/show_bug.cgi?id=497230

[2] Søren Sandmann, *Sysprof, System-wide Performance Profiler for Linux*, http://www.daimi.au.dk/~sandmann/sysprof/

[3] Paul E. McKenney, *Differential Profiling*, http://www.rdrop.com/users/paulmck/scalability/paper/profiling.2002.06.04.pdf

[4] Zhang Rui, *[RFC PATCH 0/3] perf: show package power consumption in perf*, http://lkml.org/lkml/2010/8/18/63

[3]Running Average Power Limit
[4]Model Specific Register

i915 i915_gem_object_create
skb:kfree_skb
skb:skb_copy_datagram_iovec
scsi:scsi_dispatch_cmd_error
scsi:scsi_eh_wakeup
bkl:lock_kernel
bkl:unlock_kernel
block:block_rq_complete
block:block_rq_issue
jbd2:jbd2_start_commit
jbd2:jbd2_commit_locking
ext4:ext4_allocate_inode
ext4:ext4_da_write_begin
kmem:kmalloc
kmem:kfree
power:power_start
power:power_frequency
module:module_load
module:module_put
workqueue:workqueue_insertion
workqueue:workqueue_execution
signal:signal_deliver
timer:timer_init
timer:timer_expire_entry
irq:irq_handler_entry
irq:irq_handler_exit
irq:softirq_entry
irq:softirq_exit
sched:sched_wakeup
sched:sched_switch
mce:mce_record
raw_syscalls:sys_enter
raw_syscalls:sys_exit
syscalls:sys_enter_socket
syscalls:sys_exit_socket

Table 4: Tracepoint events.